Data Structures

Queues and Lists

What is a Queue?

- A ***queue*** is an ordered collection of data items from which items may be deleted from one end (the ***front***) and and into which items may be added (the ***rear***).
- A queue has three primitive operations:
  - Empty - True if the queue's length = 0
    False if the queue's length ≠ 0
  - Insert - adds an items to the rear of the queue
  - Remove - deleted an item from the front of the queue.

Implementing A Queue

```
#include      <iostream.h>
#include      <stdlib.h>
const int     MaxQueue = 100;
class queue   {
public:
       queue(void);
       int    empty(void);
       void   insert(int x);
       int    remove(void);
private:
       int    full(void);
       void   error(char *message);
       int    items[MaxQueue];
       int    front, rear;
};
```

```
queue::queue(void)
{
       rear = 0;
       front = 1;
}

int    queue::empty(void)
{
       return(front == rear + 1);
}

int    queue::full(void)
{
       return(front == MaxQueue);
}
```

```
void   queue::insert(int x)
{
       if (full())
              error("Queue overflow");
       items[++rear] = x;
}

int    queue::remove(void)
{
       if (empty())
              error("Queue underflow");
       return(items[front++]);
}
```

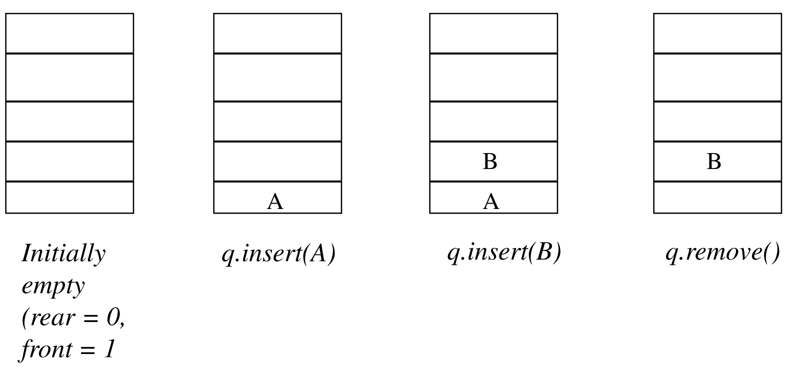
```
void   queue::error(char *message)
{
       cout << message;
       exit(1);
}
```

Initializing The Queue & `queue::empty()`

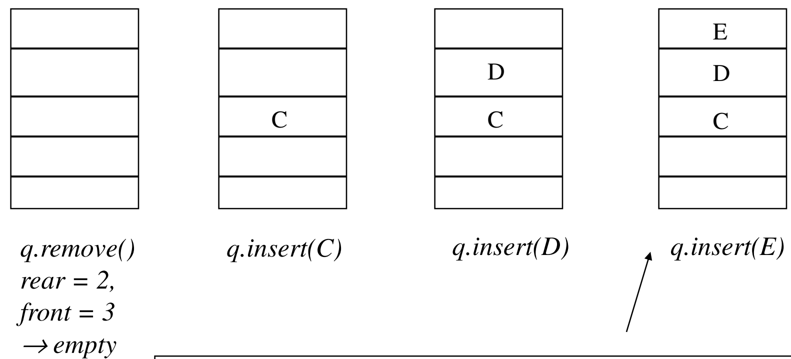
- Initially, rear is 0 and front is 1 and the queue empty.
- The number of items on the queue is :

  `rear-front+1.`

Using the Queue



Using the Queue(continued)



The queue will overflow if we add one more element
What do we do?

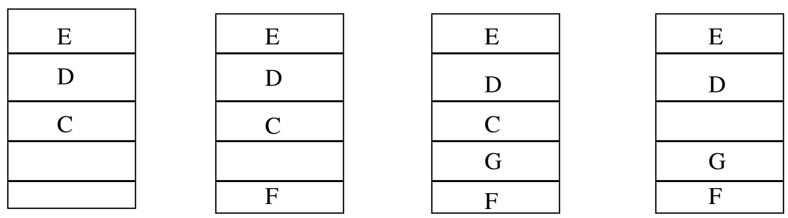
Changing the Queue Implementation

- ***Idea*** – Modify the remove to shift the queue one place toward the beginning of the array to eliminate the need for the front. But it wastes time if the array is large:

```
int    queue::remove(void)
{
       int    i, result;

       if (empty())
              error("Queue underflow");
       result = items[0];
       for  (i = 0;  i < rear - 1;  i++)
              items[i] = items[i+1];
        return(result);
}
```

Changing the Queue Implementation (continued)

- Another idea – Treat the queue like a circle



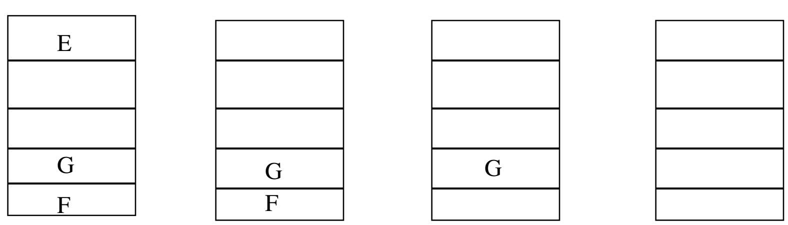
rear = 5, front = 3

q.insert(F) — *rear = 1, front = 3*

q.insert(G). — *rear = 2, front = 3*

q.remove(), — *rear = 2, front = 4*

Changing the Queue Implementation (continued)



q.remove() — *rear = 2, front = 5*

q.remove() — *rear = 2, front = 1*

q.remove(). — *rear = 2, front = 2*

q.remove(), — *rear = 2, front = 3*

<u>One problem</u> – how do we test for an empty queue?

The Final Queue Implementation

- We will have fron be the index of the element ***preceding*** the front of the queue. Now, `front == rear` implies an empty queue.
- Let's now redefine empty(), insert and remove accordingly.

The Final Queue Implementation

```
const int     MaxQueue = 100;
class queue   {
public:
       queue(void);
       int    empty(void);
       void   insert(int x);
       int    remove(void);
private:
       int    full(void);
       void   error(char *message);
       int    items[MaxQueue];
       int    front, rear;
};
```

```
queue::queue(void)
{
        rear = front = MaxQueue;
}

int     queue::empty(void)
{
        return(front == rear);
}
```

```
int     queue::full(void)
{
        int     nextrear;
        nextrear = (rear == MaxQueue)? 1 : rear + 1;
        return(front == nextrear);
}

void    queue::insert(int x)
{
        if (full())
                error("Queue overflow");
        rear = (rear == MaxQueue)? 1  : rear +1;
        items[rear] = x;
}
```

```
int     queue::remove(void)
{
        if (empty())
                error("Queue underflow");
        front = (front == MaxQueue)? 1 : front + 1;
        return(items[front]);
}

void    queue::error(char *message)
{
        cout << message;
        exit(1);
}
```

Priority Queues

- Priority queues are queues where items are added in an arbitrary order and removed in size order.
  - Ascending priority queue – smallest is first
  - Descending priority queue – largest is first
- Insert will work as before
- We need two removal procedures:
  - pqmindelete - Removes smallest item
  - pqmaxdelete – Removes largest item
- What is we need to remove an item that is in the middle?

Possible Fixes For the Priority Queue

- Mark a deleted position as empty and compact when you need the space.

  Disadvantage – wastes time and space
- Mark as empty and when inserting, use these first

  Disadvantage – each insertion becomes inefficient
- Mark as empty and compact immediately.

  Disadvantage – each compaction is time-consuming
- Maintain an ordered list

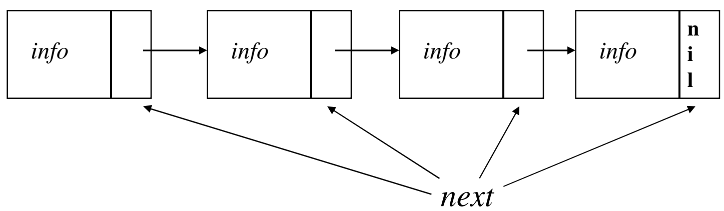
  Disadvantage – requires re-ordering for each insertion in the middle or beginning.

Possible Fixes For the Priority Queue (continued)

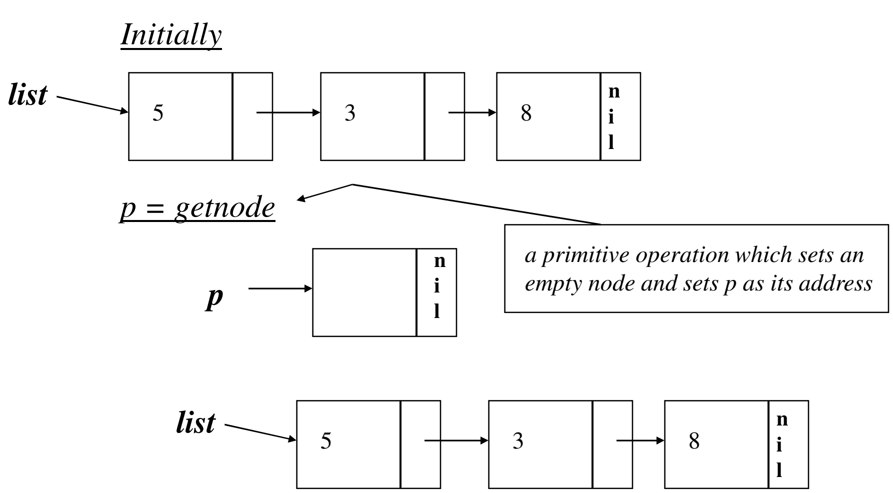
- What we have done is sequential representation of stracks and queues. It is inefficient.
- What if we need to use 2 stacks, each with their own array of 100 elements? If the first needs 25 and the second needs 125, we cannot do it.
- ***Idea*** – Use a linked list

Linked Lists

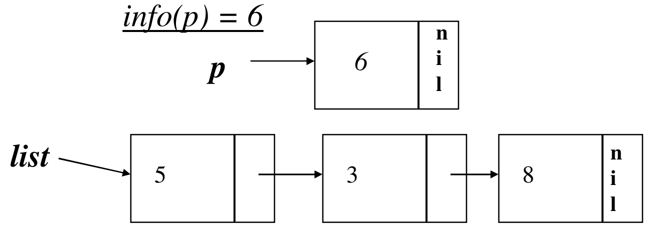
- A linked list is an array of data elemetns consisting of an information field and a next field containing the address of the next data element on the list.



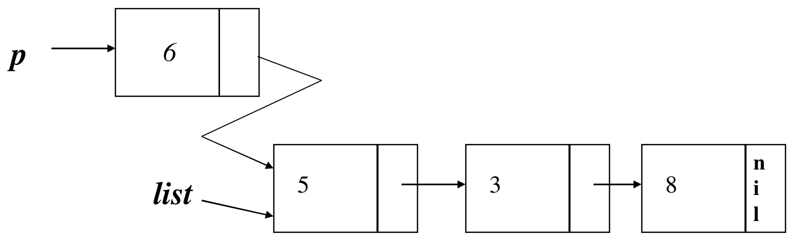
Inserting on a linked list



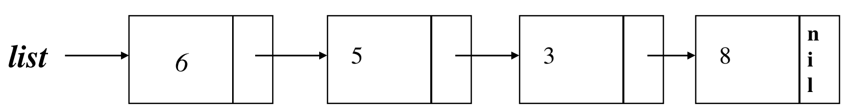
info(p) = 6



next(p) = list

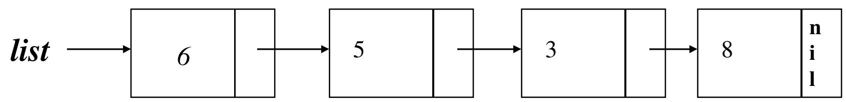


list = p

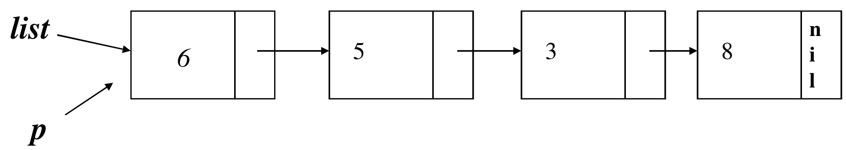


Removing From the Front of a List

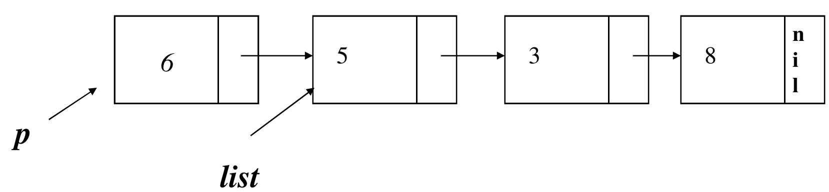
Initially



p = list

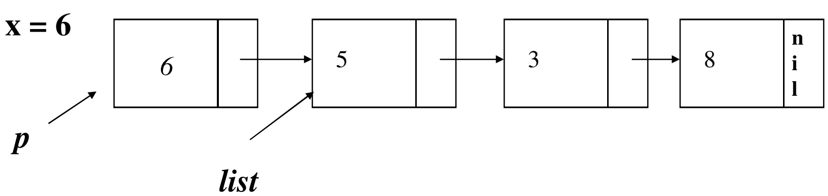


list = next(list)

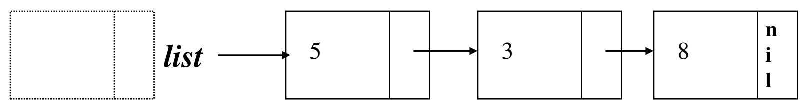


x = info(p)

x = 6



freenode(p)



x = 6

Freenode makes the node available for re-use, recycling the node.

Rewriting **`push`** and **`pop`**

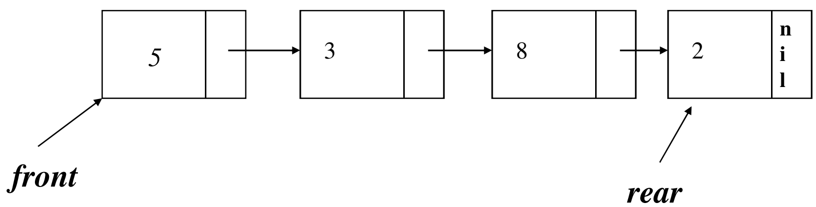
push becomes:

```
p = getnode();
info(p) = x;
next(p) = s;
s := p;
```

pop becomes:

```
if (empty(s))
  error(“Stack underflow”);
p = s;
s = next(p);
x = info(p;
freenode(p);
```

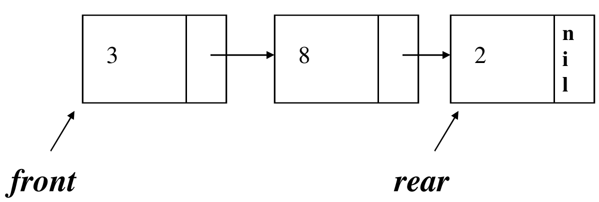
Linked List-Implementation of Queues



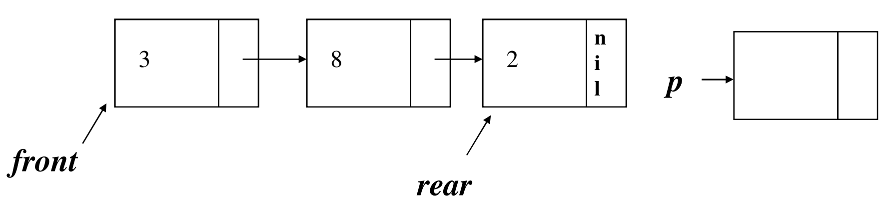
Remove works in the exact same fashion as pop

Inserting onto a Queue

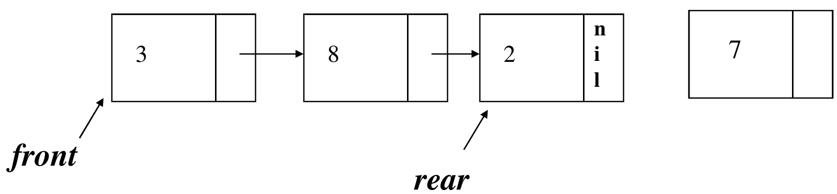
Initially



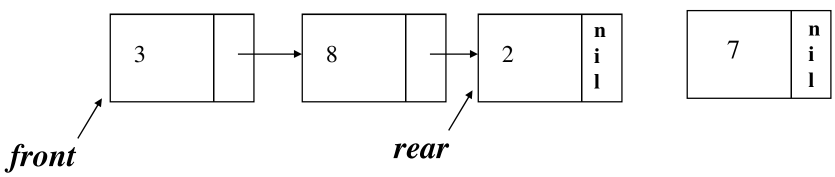
p = getnode



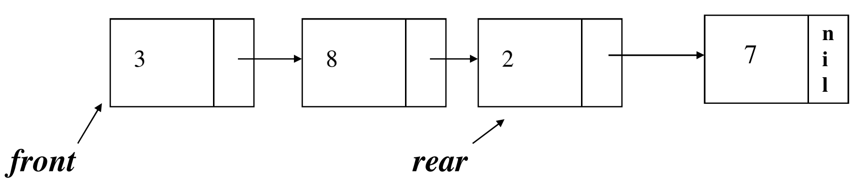
info(p) = x



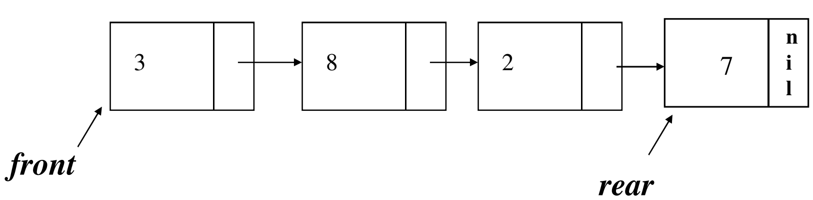
next(p) = nil



next(rear) = p



rear = p



Queue Operations

insert()

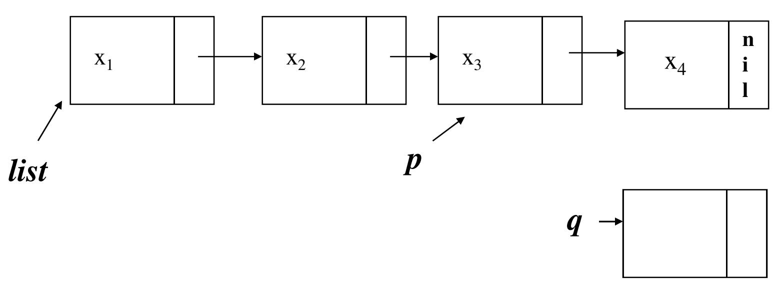
```
p = getnode
info(p) = x
next(p) = nil
next(rear) = p
rear = p
```

remove()

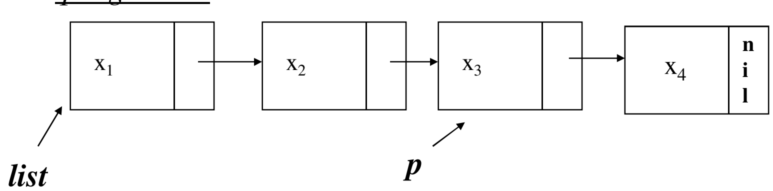
```
if (empty(q))
      error (“Queue underflow”)
p = front
x = info(p)
front = next(p)
if (front = nil)
      rear = nil
freenode(p)
return(x);
```

Linked List Operations - **`insafter`**

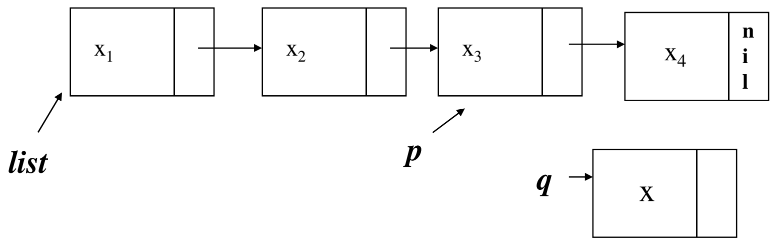
Initially



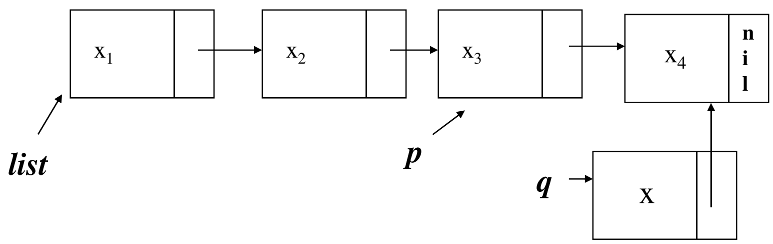
q = getnode



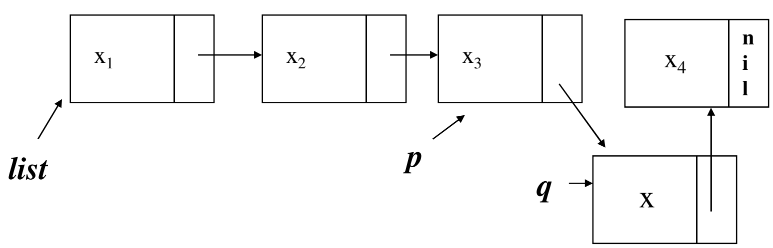
info(q) = x



next(q) = next(p)

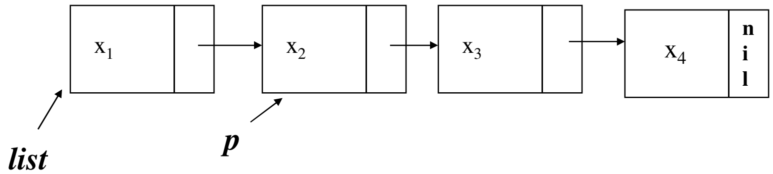


next(p) = q

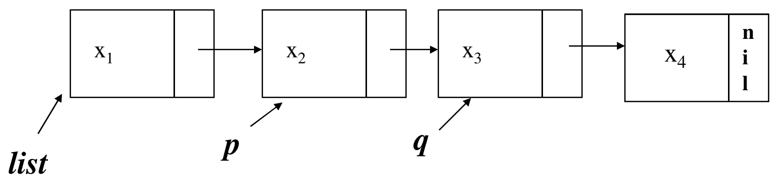


Linked List Operations - **`delafter`**

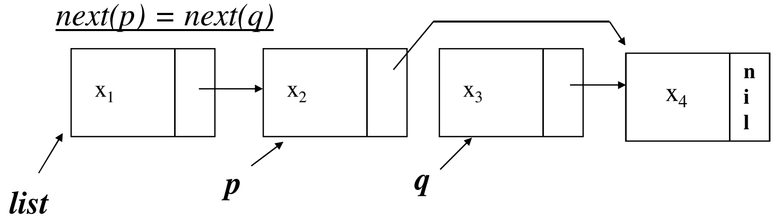
Initially



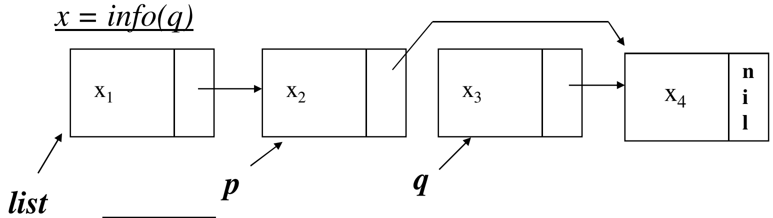
q = next(p)



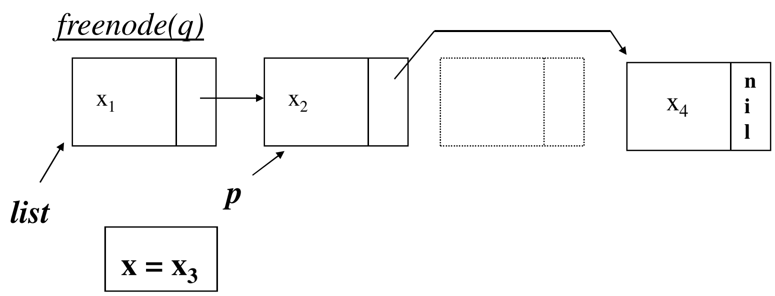
next(p) = next(q)



x = info(q)



$x = x_3$



List Operations

delafter()

```
q = next(p)
x = info(p)
next(p) = next(q)
freenode(q)
```

insafter()

```
q = getnode
info(p) = x
next(q) = next(p)
next(p) = q
```

Priority Queue Implementation

We can easily add an element in its sorted order on the list:

```
q = nil;
p = list;
while (p != nil && x > info(p)      {
      q = p;
      p = next(p);
}
if (q = nil)
      push(list, x);
else
      insafter(list, x);
```

We can use **`delafter(rear)`** or **`pop()`** to remove an item and create an ascending or descending priority queue.

Implementing Linked Lists in C++

```
const int   numnodes = 500;   // Maximum # of nodes
enum        boolean     {false, true};

typedef     struct      {
      int   info, next;
} nodetype;
```

```
class  list  {
public:
       list(void);
       int    getnode(void);
       void   setnode(int p, int x);
       void   freenode(int p);
       void   insafter(int p, int x);
       int    delafter(int p);
inline int    getvalue(int p) {return(node[p].info);}
inline int    getnext(int p) {return(node[p].next);}
protected:
       void   error(char *message);
       int           avail;
       nodetype      node[numnodes];

};
```

```
list::list(void)
{
       int    i;
       avail = 0;
       for  (i = 0;  i < numnodes-1; i++)
              node[i].next = i+1;
       node[numnodes-1].next = -1;
}
// error() -Prints an error message and terminates
void   list::error(char *message)
{
       cerr << message << endl;
       exit(1);
}
```

```
int   list::getnode(void)
{
      int   p;

      if (avail == -1)
            error("List overflow");

      p = avail;
      avail = node[avail].next;
      node[p].next = -1;
      return(p);
}
```

```
void  list::setnode(int p, int x)
{
      node[p].info = x;
}

void  list::freenode(int p)
{
      node[p].next = avail;
      avail = p;
}
```

```
void  list::insafter(int p, int x)
{
      int   q;

      if (p == -1)       {
            cout << "Void insertion" << endl;
            return;
        }
      q = getnode();
      node[q].info = x;
      node[q].next = node[p].next;
      node[p].next = q;
}
```

```
int   list::delafter(int p)
{
      int   q, x;
      if ((p == -1) || (node[p].next == -1)) {
            cout << "Void deletion" << endl;
            return;
      }
      q = node[p].next;
      x = node[q].info;
      node[p].next = node[q].next;
      freenode(q);
      return(x);
}
```

Using an Array to Implement a Linked List

	info	next
avail= 0		11
1	26	-1
list = 2	11	3
3	5	6
list = 4	13	5
5	35	10
6	6	7
7	17	8
8	41	14
9		12
10	86	16
11		9
12		13
13		17
14	45	1
15	31	-1
16	22	15
17		-1

Pointers

- If we write:

```
int          num1;
```

num1 refers to the value stored at a given address(the address is supplied by the compiler).

- If we write:

```
int          *num1;
```

num2 is a pointer provides an address that will be used to store an integer.

- ***num2** is the value stored at **num2** and can be used like **num1**. However, this is NOT the common use.

Pointers and Records

```
typedef    struct   {
     char name[20];
     int  empnumber;
     floatsalary;
}    workerrec;
typedef        workerrec *workerptr;
workerptr worker;
```

*Worker is an address at the beginning of the record. *worker is the record itself and is used like other record identifiers.*

Pointers and Records(continued)

- We could write:

  `(*worker).empnumber`

  or

  `worker -> empnumber`

  The second is the preferred form.

new

- Pointers have to be allocated and freed, i.e., the memory is given to the program and taken away when it is no longer needed.
- If we declare a pointer nameptr

```
name    *nameptr;
```

- we can allocate a pointer by writing:

```
nameptr = new name
```

- In our earlier example, we would allocate storage by writing:

```
worker = new workerrec;
```

- We can assign a value to the employee # by writing:

  worker -> empnumber = 123;

delete

- When we are ready to free the memory used by our data we write:

```
delete nameptr;
```

- In our example it becomes:

```
delete worker;
```

- If we allocate a new pointer by writing:

```
worker = new workerrec;
```

- The data stored previously is gone.

Using Pointers to Create Linked Lists

```
struct node   {
       int    info;
       struct node *next;
};
typedef       struct node *NodePtr;
NodePtr       getnode(void)
{
       NodePtr       p;
       p = new struct node;
       return(p);
}
void   freenode(NodePtr p)
{
       delete p;
}
```

Using Pointers To Implement Stacks – **`push()`**

```
void   push(NodePtr &list, int x)
{
       NodePtr      p;
       p = getnode();
       p -> info = x;
       p -> next = list;
       list = p;
}
```

Using Pointers To Implement Stacks: **pop()**

```
int    pop(NodePtr &list)
{
       NodePtr      p;
       int    x;
       if (list == NULL)    {
              cerr << "Popping an empty list\n";
              exit(1);
       }
       p = list;
       list = list -> next;
       x = p -> info;
       freenode(p);
       return(x);
}
```

Using Pointers To Implement Queues

```
typedef struct node     *NodePtr;
struct queue      {
      NodePtr     front, rear;
};

int   empty(struct queue *q)
{
      return(q -> front == NULL);
}
```

Other Linked List Procedures

- There are other procedures that apply to linked lists.
- They include:
  - **insafter** – Insert after node p
  - **delafter** – Delete the node after node p
  - **place** – Place information in order on the list
  - **insend** - Insert at the end of the list
  - **search** – Search the list for a specific datum

Using Pointers To Implement Queues – `insert()`

```
void   insert(struct queue *q, int x)
{
       NodePtr      p;

       p = getnode();
       p -> info = x;
       p -> next = NULL;
       if (q -> rear == NULL)
              q -> front = p;
       else
              (q -> rear) -> next = p;
        q -> rear = p;
}
```

Using Pointers To Implement Queues – `remove()`

```
int     remove(struct queue *q)
{       NodePtr       p;
        int    x;
        if (empty(q)) {
              cerr << "Queue underflow" << endl;
              exit(1);
        }
        p = q-> front;
        x = p -> info;
        q -> front = p -> next;
        if (q -> front == NULL)
              q -> rear = NULL;
        freenode(p);
        return(x);
}
```

`insafter()` – Inserting after node p

```
void insafter(NodePtr p, int x)
{
        NodePtr       q;

        if (p == NULL)      {
              cerr << "Void insertion" << endl;
                 exit(1);
        }
        q = getnode();
        q -> info = x;
        q -> next = p -> next;
        p -> next = q;
}
```

`delafter()` – Deleting the node after p

```
int     delafter(NodePtr p)
{
        NodePtr        q;
        int     x;
        if ((p == NULL) || (p -> next == NULL)) {
                cerr << "Void deletion" << endl;
                exit(1);
        }
        q = p -> next;
        x = q -> info;
        p -> next = q -> next;
        freenode(p);
        return(x);
}
```

`place()` – Placing Datum in Order

```
void    place(NodePtr &list, int x)
{
        NodePtr        p, q;

        q = NULL;
        for (p = list;  p != NULL && x > p -> info;
                        p = p-> next)
                q = p;
        if (q == NULL)
                push(list, x);
        else
                insafter(q, x);
}
```

insend() – Inserting At the End of the List

```
void   insend(NodePtr &list, int x)
{      NodePtr       p, q;
       p = getnode();
       p -> info = x;
       p -> next = NULL;
       if (list == NULL)
              list = p;
       else   {
              //Search for the end of the list
              for  (q = list;  q -> next != NULL;  q = q
-> next)
                     ;
              q -> next = p;
        }
}
```

search() – Searching the List for a Datum

```
NodePtr       search(NodePtr list, int x)
{
       NodePtr       p;

       for (p = list;  p != NULL; p = p -> next)
              if (p -> info == x)
                     return(p);
       // x is not in the list
       return(NULL);
}
```

Header Nodes

- Header Node – the first may have general information pertaining to the list.
- E.g., If inventory, the part numbers of the components.

Bank Simulation

- The bank program is an example **event-driven simulation**.
- We keep track of an event list with arrival nodes and departure nodes to account for customers entering and exiting the bank.
- We wish to determine the average duration time.

The Bank Program

```
#include    <iostream.h>
#include    <stdlib.h>

const int   numlines = 4;
struct node {
      int         duration, time, type;
      struct node *next;
};

typedef     struct node *NodePtr;
```

```
int         empty(NodePtr p);
NodePtr     getnode(void);
void        freenode(NodePtr p);
void        error(char *message);

typedef  struct   {
      NodePtr     front, rear;
      int         num;
}     queue;
```

```
class bank  {
public:
      bank(void);
      int         getshortest(void);
      inline int  getnumber(int qindx)
                        {return(q[qindx].num); }
      inline NodePtr    getfront(int qindx)
                        {return(q[qindx].front);}
      void        insert(int qindx, NodePtr info);
      struct node remove(int qindx);
private:
      queue q[numlines];
};
```

```
bank::bank(void)
{
      int   qindx;

      for  (qindx = 0;  qindx < numlines;  qindx++)
      {
            q[qindx].num = 0;
            q[qindx].front = NULL;
            q[qindx].rear = NULL;
        }
}
```

```
int    bank::getshortest(void)
{
       int    i, j, small;

       j = 0;
       small = q[0].num;
       for (i = 1; i < 4; i++)
              if (q[i].num < small)      {
                     small = q[i].num;
                     j = i;
              }
        return(j);
}
```

```
void   bank::insert(int qindx, NodePtr info)
{
       NodePtr       p;

       p = getnode();
       *p = *info;
       p -> next = NULL;
       if (q[qindx].rear == NULL)
              q[qindx].front = p;
       else
              (q[qindx].rear) -> next = p;
       q[qindx].rear = p;
       q[qindx].num++;
}
```

```
struct node  bank::remove(int qindx)
{      NodePtr      p;
       struct node  info;

       if (q[qindx].front == NULL)
              error("Queue underflow");
       p = q[qindx].front;
       info = *p;
       q[qindx].front = p -> next;
       if (q[qindx].front == NULL)
              q[qindx].rear = NULL;
       delete(p);
       --q[qindx].num;
       return(info);
}
```

```
class eventlist   {
public:
       eventlist(void);
       void        place(struct node *info);
       inline int  emptylist(void)
                   {return(evlist == NULL); }
       struct node       pop(void);
       void  push(struct node *info);
       void  insafter(NodePtr p, struct node *info);
private:
       NodePtr     evlist;
};
```

```
eventlist::eventlist(void)
{
       evlist = NULL;
}



void   eventlist::push(NodePtr x)
{
       NodePtr      p;
       p = getnode();
       *p  = *x;
       p -> next = evlist;
        evlist = p;
}
```

```
struct node   eventlist::pop(void)
{
       NodePtr             p;
       struct node   x;

       if (evlist == NULL)
              error("ERROR! the list is empty.");
       p = evlist;
       evlist = p -> next;
       x = *p;
       x.next = NULL;
       freenode(p);
       return (x);
}
```

```
void  eventlist::insafter(NodePtr p,
                          struct node *info)
{
      NodePtr     q;

      if (p == NULL)
            error("Void insertion");

      q = getnode();
      *q = *info;
      q -> next = p -> next;
      p -> next = q;
}
```

```
void  eventlist::place(struct node *info)
{
      NodePtr      p, q;

      q = NULL;
      for (p = evlist;
            p != NULL && info -> time > p -> time;
            p = p -> next)
                  q = p;

      if (q == NULL)
            push(info);
      else
            insafter(q, info);
}
```

```
void arrive(int time, int dur);
void depart(int qindx, int dtime,
          float &tottime, float &count);


eventlist event;
bank      mybank;
```

```
int   main(void)
{
      int   atime, dtime, dur, qindx, x;
      float count, tottime;
      struct node auxinfo;


      //Initializations
      count = 0;
      tottime = 0;
```

```
//Initialize the Event List with the first arrival
cout << "Enter time and duration\t->";
cin >> auxinfo.time >> auxinfo.duration;

auxinfo.type = -1;  // An arrival
event.place(&auxinfo);

x = event.emptylist();
while (!x)    {
       auxinfo = event.pop();
       //Check if the next even is an arrival
       //or a departure
```

```
       if (auxinfo.type == -1)    {
              // An arrival
              atime = auxinfo.time;
              dur = auxinfo.duration;
                  arrive(atime, dur);
       }
       else   {
              // A departure
              qindx = auxinfo.type;
              dtime = auxinfo.time;
              depart(qindx, dtime, tottime, count);
          }
       x = event.emptylist();
}
```

```
        if (count != 0)
                cout << "The average time is “
                        << tottime/count << endl;
        else
                cout << "There were no transactions to”
                        << ” average\n";
         return(0);
}

void    error(char    *message)
{
        cout << message << endl;
         exit(1);
}
```

```
void    arrive(int atime, int dur)
{
        int             shortest;
        struct node     auxinfo;
        //Find the shortest queue
        shortest = mybank.getshortest();
        //Shortest is the shortest queue.
        //Insert a new customer
        auxinfo.time = atime;
        auxinfo.duration = dur;
        auxinfo.type = shortest;
        mybank.insert(shortest, &auxinfo);
```

```
// Check if this is the only node on the queue.
// If it is, the customer's departure node must
// be placed on the event list.
if (/*mybank.getnumber(*/shortest/*)*/  == 1)  {
        auxinfo.time = atime + dur;
        event.place(&auxinfo);
}

// If any input remains, read the next data pair
// and place an arrival on the event list
cout << "Enter the time\t->";
cin >> auxinfo.time;
```

```
        if (auxinfo.time >= 0)     {
                cout << "Enter duration\t->";
                cin >> auxinfo.duration;
                auxinfo.type = -1;
                event.place(&auxinfo);
        }
}
```

```
void   depart(int qindx, int dtime, float &tottime,
              float &count)
{
       NodePtr              p;
       struct node   auxinfo;

       auxinfo = mybank.remove(qindx);
       tottime += dtime - auxinfo.time;
       count++;
       // If there are any more customers on the queue
       // place the departure of the next customer onto
       // the event list after computing its departure
       // time
```

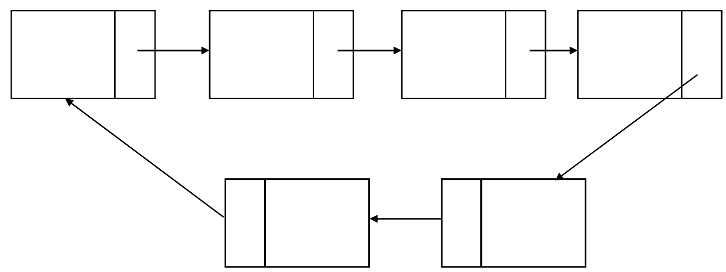
```
       if (mybank.getnumber(qindx) > 0)  {
              p = mybank.getfront(qindx);
              auxinfo.time = dtime + p -> duration;
              auxinfo.type = qindx;
              event.place(&auxinfo);
       }
}
```

```
NodePtr getnode(void)
{
        NodePtr             p;

        p = new struct node;
        return(p);
}

void    freenode(NodePtr p)
{
        delete(p);
}
```

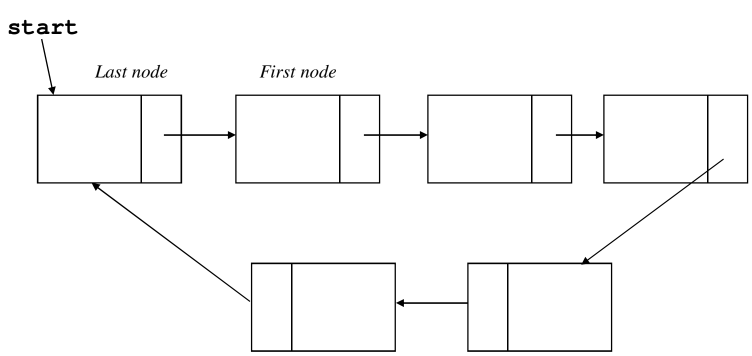
Circular List

- A circular list has the **`next`** field in the last node point to the first node in the list.
- Since a circular list has no obvious beginning and end, we establish this by convention by having the **`start`** pointer point to the last node in the list.

A Circular List



First and Last Nodes on a Circular List



CircularList.h

```
#ifndef__CLIST__
#define__CLIST__

#include<iostream>
#include<stdlib.h>
#include<string.h>

using namespace std;
#endif
struct node{
      int   info;
      struct node *next;
};
```

```
typedef     struct node *NodePtr;

class  CircularList{
public:
      CircularList(void);
      inline boolempty(void)
            {return (start == NULL);}
      void  push(int x);
      int   pop(void);
      void  insert(int x);
      int   remove(void);
      void  insafter(NodePtr p, int x);
      int   delafter(NodePtr p);
      NodePtr     getnode(void);
      void        setnode(NodePtr p, int x);
      void        freenode(NodePtr p);
```

```
      inline int getvalue(NodePtr p)
                  {return(p->info);}
      inline NodePtr getnext(NodePtr p)
                   {return(p->next);}
private:
      void  error(char *message);
      NodePtr     start;
};
```

Circular.cpp

```
#include "CircularList.h"

CircularList::CircularList(void)    {
      start = NULL;
}

NodePtr     CircularList::getnode(void)   {
      NodePtr p;

      p = new struct node;
      return p;
}
```

```
void  CircularList::setnode(NodePtr p, int x)   {
      p ->info = x;
}

void  CircularList::freenode(NodePtr p)   {
      delete p;
}

void  CircularList::push(int x)     {
      NodePtr     p;
      p = getnode();
      p -> info = x;
      if (empty())
            start = p;
      else
            p -> next = start -> next;
      start -> next = p;
}
```

```
int         CircularList::pop(void) {
      int   x;
      NodePtr     p;

      if (empty())
            error("Stack underflow");

      p = start -> next;
      x = p -> info;
      if (p == start)
            start = NULL;
      else
            start -> next = p -> next;
      freenode(p);
      return (x);
}
```

```
void  CircularList::insert(int x)   {
      NodePtr     p;
      p = getnode();
      p ->info = x;
      if (empty())
            start = p;
      else
            p->next = start->next;
      start -> next = p;
      start = p;
}
```

```
int         CircularList::remove(void)    {
      int   x;
      NodePtr     p;

      if (empty())
            error("Stack underflow");

      p = start -> next;
      x = p -> info;
      if (p == start)
            start = NULL;
      else
            start -> next = p -> next;
      freenode(p);
      return (x);
}
```

```
void  CircularList::insafter(NodePtr p, int x)  {
      NodePtr     q;

      if (p == NULL)    {
            cerr << "Void insertion" << endl;
                exit(1);
      }
      q = getnode();
      q -> info = x;
      q -> next = p -> next;
      p -> next = q;
}
```

```
int   CircularList::delafter(NodePtr p)   {
      NodePtr     q;
      int         x;

      if ((p == NULL) || (p == p ->next))
            error("Void deletion");

      q = p ->next;
      x= q -> info;
      p -> next = q -> next;
      freenode(q);
      return(x);
}
```

```
void  CircularList::error(char *message)  {
      cerr << message << endl;
      exit(1);
}
```

The Josephus Program

```
#include    "CircularList.h"

void josephus(void);

int   main(void)  {
      josephus();
      return(0);
}
```

```
void josephus(void)     {
  char      *end = "end";
  char name[MaxLen];
  int i, n;
  CircularList cl;

  cout << "Enter n\t?";
  cin >> n;

  // Read the names placing each
  // at the end of the list
  cout << "Enter names:\n";
  cin >> name;
```

```
  // Form the list
  while (strcmp(name, end) != 0) {
    cl.insert(name);
    cin >> name;
  }

  cout << "The order in which the "
            << "soldiers were eliminated:"
            << endl;

  // Continue counting as long as more
  // than one node remains on the list
  while (cl.getStart()
                != cl.getnext(cl.getStart())) {
```

```
    for (i = 1; i < n;  i++)
      cl.setnext(cl.getnext(cl.getStart()));

    // start -> next points to the nth node
    cl.delafter(cl.getStart(), name);
    cout << name << endl;
  }

  // Print the only surviving name
  cl.getInfo(cl.getStart(), name);
  cout << "The soldier who escapes is: "
            << name << endl;
  cl.freenode(cl.getStart());
  cout << "All done" << endl;
}
```