Linked implementations of Stacks and Queues

SE202 Data Structures

New

New is a mechanism in C++ for allocating memory to objects of a given type

```
new Node
```

creates a new object of type Node and returns a pointer to that object.

```
new node(parameters)
```

creates an object of type T

Delete

If p points to an object created via use of the new operator, then the statement

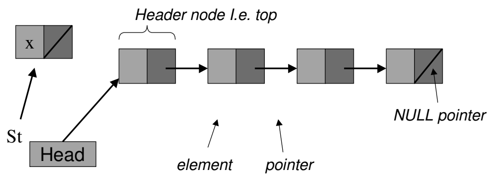
```
delete p;
```

deallocates the object to which p was pointing. If the type has a destructor, the destructor is invoked prior to de-allocation.

Stacks: Inserting & Deleting

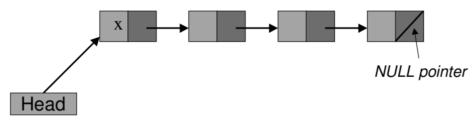
- Pushing an element on a stack is similar to adding an element to the front of a linked list.
- A stack can be accessed through the top element and a linked list representation can be accessed through the pointer to the first element in the list. This element represents the last element inserted into the stack and hence the top of the stack.
- Removing the first element of the list is analogous to popping the stack.

push x on the stack



If stack is empty put head/top pointing at node to be added
else link node to be added to existing stack and move head/top
to point at it.
Don't forget to increment the size of the stack if keeping a record!

Pop the stack

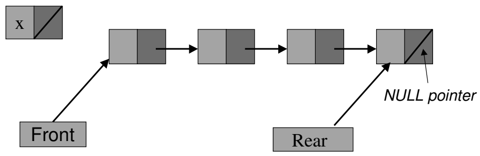


Move head to head -> next
Decrement size
Free the node to de-allocate memory

Stacks:Top, IsEmpty, Size

- To find out what is at the top of the stack just return the value of head -> x
- The stack is empty if head is pointing at NULL
- How do I determine the size of the stack?
  - `Keep a counter which increments when I add and decrements when I delete a node`
- Don't forget about Underflow!
- Do we need to check for Overflow?

Queues



A Queue is a FIFO Structure: Elements are inserted at the rear and removed from the front.

Queue Operations

- Empty and remove are as in the linked list representation of a stack.
- However, care must be taken when removing the last element: rear must be set to Null.
- Elements are inserted at the rear of the queue


```
#include <stdio.h>


 // This implementation uses a linked list
//  to implement the specification of the
 // queue.

typedef struct node{
                int dta;
                struct node *next;
                } IntNode;
```


Initialise Queue

```
 class IntQueue{
        IntNode *queue;
        IntNode *tail;
        int size;
        public:
        IntQueue(){
                queue = NULL;
                tail = NULL;
                size =  0;
                }
}
```


Insert in Queue

```
void Join(int x){
    IntNode *t;
    t =  new IntNode;
    t->dta = x;
    t->next = NULL;
    size = size + 1;
```

```
if (queue == NULL)
        {queue = t;
         tail = t;
        }
        else
        {
         tail->next = t;
         tail = t;
        }

}
```


Remove from Queue

```
void Leave(){ IntNode *t;
              t = queue;
              queue = queue->next;
              size = size - 1;
              delete t;
              t= NULL;
            }
```


Other Queue Methods

```
 int Head(){ return queue->dta; }
 int Size(){ return size; }
 int IsEmpty(){ if(size == 0)
               return 1;
               else
               return 0;
             }
 void Clear();  // Clear requires a loop.
// Therefore, cannot be defined in-line
};
```



```
 // external definiton of Clear follows:

void IntQueue :: Clear()
{
          IntNode *t;
          while(queue != NULL)
          { t = queue;
            queue = queue->next;
            delete t;
            t= NULL;
          }
}
```



```
 void main()
{
        IntQueue qu1;
        int i = 0;
        while(i < 20)
         {
           qu1.Join(i);
           i = i + 1;
         }
         while (qu1.IsEmpty() != 1)
          { cout << qu1.Head();
            qu1.Leave();
          }
}
```


Priority Queues as Linked Lists

- An ordered linked list can be used to represent a priority queue. This requires examining an average of n / 2 nodes for insertion and one node for deletion.
- An unordered representation requires the examination of one node for insertion and n nodes for deletion.
- Advantage of linked implementation over array implementation is that no movement of elements is required when an item is deleted.

Circular Lists

- Given a pointer p to a node in a linear list, we cannot reach any of the nodes that precede the node pointed to by p.
- A circular list has a pointer pointing from the last node to the front node rather than to NULL.
- From any point in such a list it is possible to reach any other point in the list.
- By convention we set rear to be the node p is pointing to and front to be p->next.

- This convention allows us to add / remove an element from either the front or the rear of the list.
- The empty circular list is represented by a NULL pointer

Stacks as Circular Lists

- Let Stack be a pointer to the last node of a circular list
- Let the first node represent the top of the stack
- An empty stack is represented by the null list

Queues as a circular list

- A linear linked list is specified by two pointers, one to the front and one to the rear of the list.
- Using a circular list a queue may be specified by a single pointer to the list.
- This pointer points to the rear of the list and the pointer -> next points to the front.

The Josephus Problem

- A group of m soldiers are surrounded by the enemy and there is only a single horse for escape.
- The soldiers determine a pact to see who will escape and summon help.
- The form a circle and pick a number n which is between 1 and m.
- One of their names is also selected at random.

- Beginning with the soldier whose name was picked they begin counting clockwise around the circle.
- When the count reaches n that soldier is removed from the circle and the count begins with the next soldier.
- The process continues until 1 soldier is left. This soldier takes the horse and summons help.

The problem!

- Given n, m, the ordering of the soldiers in the circle and the name of the soldier where counting commences, determine the order in which the soldiers are eliminated and which soldier escapes.

- Clearly, a circular list in which each node represents a soldier is a natural data structure for solving this problem.

Doubly Linked Lists

Drawbacks of a circular linked list

- can’t traverse a list backwards
- can’t delete an element from a circular list given only a pointer to that node

In a DLL each node has 2 pointers, one to its predecessor and one to its successor.

```
struct node{
 int info;
 struct node *left, *right;
};
typedef struct node *nodeptr;
```