Priority Queues I: Stacks and Queues

A *priority queue* is a data structure Q which contains finitely many *items*, each of which has a *priority*, and hs the following operators.

1. Initialize Q to be empty.
2. Insert an arbitrary item into Q.
3. Delete the item of highest priority from Q.
4. Determine whether Q is empty.

There are three standard kinds of priority queues, stacks, queues, and heaps. These are distinguised by their definition of priority. Priority can be defined in these three ways.

1. The most recently inserted item has priority, in which case the priority queue is called a *stack*. In this case, insert is call *push* and deletion is called *pop*. Originally, this structure was called a *push-down stack*.
2. The least recently inserted item has priority, in which case the priority queue is called a *queue*. Insert is called *enqueue* and deletion is called *dequeue*.
3. Data have an instrinsic priority measure, such as larger or smaller, in which case the priority queue is called a *heap*. If the larger datum has priority, it is called a *max-heap*, if smaller, *min-heap*. Implementation is the same, regardless of the definition of priority.

For simplicity, we shall assume that items have type integer.

Implementation of Stack

Array Implementation

The items are stored in an array in the order they were inserted. When an item is inserted (pushed) it appended to the array, and when an item is deleted (popped) the end item is returned. We could implement stack as a structured type in C++. N must be at least as large as the maximum number of items the stack will hold.

```
struct stak
 {
  int item[N];
  int size; // the number of items in the stack
 };
```

(Why is the word “stack” misspelled?)

Here are C++ implementations of the operators of stack.

```
void init(stak&S)
 {
  S.size = 0;
 }

bool empty(stak S)
 {
  return S.size == 0;
 }

void push(stak&S, int newitem)
 {
  S.item[S.size] = newitem;
  S.size++;
 }

int pop(stak&S)
 {
  assert(not empty(S));
  S.size --;
  return S.item[S.size];
 }
```

Linked List Implementation of Stack

The items are stored in a linked list, where each node of the list contains one item. The front of the list is the top of the stack, where items are inserted and deleted. We let "stak" be a pointer type to a node. We let the overall structure be of type stak, which is NULL if the list is empty.

```
struct staknode;
typedef staknode*stak;
struct staknode
 {
  int item;
  stak next;
 };
```

Here are C++ implementations of the operators.

```
void init(stak&S)
 {
  S = NULL;
 }
```

```
bool empty(stak S)
 {
  return S == NULL;
 }

void push(stak&S, int newitem)
 {
  stak top = new staknode;
  top->item = newitem;
  top->next = S;
  S = top;
 }

int pop(stak&S)
 {
  assert(not empty(S));
  int rslt = S->item;
  S = S->next;
  return rslt;
 }
```

Implementation of Queue

Array Implenetation

We define a new type, postn, which is simply int with a different name, to avoid confusion.

```
int const N = 20;

typedef int postn;
struct queue
 {
  int item[N]; // How large must N be?
  postn front; // a position in the array, which is an integer
  postn rear; // a position in the array, which is an integer
 };
```

Here are C++ implementations of the operators of queue.

```
void init(queue&Q)
 {
  Q.front = 0;
  Q.rear = 0;
 }
```

```
bool empty(queue Q)
 {
  return Q.front == Q.rear;
 }

void enqueue(queue&Q, int newitem)
 {
  assert(Q.rear < N);
  Q.item[Q.rear] = newitem;
  Q.rear++;
 }

int dequeue(queue&Q)
 {
  assert(not empty(Q));
  int rslt = Q.item[Q.front];
  Q.front++;
  return rslt;
 }
```

0.0.1 Linked List Implementation of Queue

This implementation is considerably more involved than the previous three implementations, which are fairly straightforward. A standard method is to have a linked list, each node of which contains an item of the queue, and pointers to both the front and rear nodes are need to access the list. This method is inelegant, since you need two pointers for each queue, and the impelentation code is different if the queue becomes empty. In the implementation below, we use a circular linked list with a dummy node. Thus, if the queue has n items, the structure has $n + 1$ nodes. The rear node points to the dummy node, which points to the front node. The structure is accessed by a single pointer to the dummy node. None of the operations require different code for special cases. Insertion of a new item is done by writing the new item into the dummy node and creating a new dummy node. Here is the code.

```
struct queuenode;
typedef queuenode*queue;
struct queuenode
 {
  int item;
  queue next;
 };

void init(queue&Q)
 {
  Q = new queuenode;  // the dummy node
```

```
  Q->next = Q;  // new dummy is set to point to itself.
 }

bool empty(queue Q)
 {
  return Q->next == Q; // if dummy points to itself return true, else false
 }

void enqueue(queue&Q, int newitem)
 {
  queue temp = new queuenode; // the new dummy node
  temp->next = Q->next;
  Q->item = newitem; // the new item is written to the old dummy node.
  Q->next = temp;  // the old dummy node is the new rear
                   // which points to the new dummy.
  Q = temp; // Q now points to the new dummy node.
 }

int dequeue(queue&Q)
 {
  assert(not empty(Q));
  int rslt = Q->next->item; // save the contents of the front node
  Q->next = Q->next->next; // delete the front node
  return rslt; // the saved item of the old front node
 }
```