Priority queues and sorting

Priority queues
Compositions
Insertion sort
Selection sort
Heap sort

Read Chapter 18 of the textbook!

Priority queues

- A priority queue is an ADT for storing a collection of elements
- The elements may be arbitrarily inserted but removal is in order of priority
  - priority is specified using a parameter which is defined for each element stored in the collection
  - the parameter used to implement priority is called the **priority** or **key** for the element
  - priorities are not unique, and may be changed if necessary
  - elements with the highest priority will be removed first
  - there may be multiple elements with the same priority
  - in which case it is common to form a FIFO queue of those elements

Priority queues

- Example: Waiting queue for an event
- Suppose the organisers of an event will sell tickets following a priority queue, aiming at VIP attendees.
  - The event is sponsored by company X, which has a loyalty program.
  - Clients who call to express interest for the event are put in a waiting queue. When sales start, they will be called by order of priority.
  - The priority follows the number of loyalty points accumulated (not the time that they called and were put in the queue).
  - If two customers have the same number of loyalty points, the tie is broken by the time of the call.

- How would you implement that?
  - Which data structure would you use?
  - What would be the object stored composed of?
- Think about it for 5 minutes.

Priority queues

- A priority queue is a container for which every element has a priority, assigned at the time of insertion, that provides a means of selecting the next element to be removed
- A priority queue PQueue extends the Queue class by adding the following fundamental method:

```
// insert element elem with priority k into PQueue
void enqueue(Key k, value_type elem)
```

- Of course, `dequeue` will also change, so that when it is called, the element with the highest priority is removed
- Note that only the element should be returned, not its priority (unless that is an explicit requirement)

Using priority queues for sorting

- A set of *n* elements can be sorted by inserting them into a priority queue *P* and then calling `dequeue()` until *P* is empty
- The algorithm is:
  - put the *n* elements of the set into an empty priority queue using *n* `enqueue` operations
  - extract the elements from *P* using *n* `dequeue` operations
- For example, if the elements were stored in a sequence, they could be removed from the sequence into the priority queue, then added back to the sequence, thus sorting the elements in the sequence
- This is the underlying method used by popular sorting algorithms such as selection sort, insertion sort and heap sort

Priority queue ADT

- Using a linked list as the underlying data structure

```
// Adds element to the tail of the queue with a default priority
void enqueue(value_type elem)

// Adds element to the tail of the queue with specified priority
void enqueue(int priority, value_type elem)

// Removes the element with the highest priority
value_type dequeue()
```

- Now, how do we keep the element and its priority connected?
- Use a composition pattern

Priority queue ADT

- The composition pattern defines a single object *c*, which is the composition of other objects
- The objects stored in the queue will be pairs (key, element) implemented as a new class

```
template <typename value_type>
class Comp
{
public:
    Comp(Key k, value_type e) {itemKey = k; elem = e;} // Constructor

    // Query member functions
    Key get_key() {return itemKey;}
    value_type get_element() {return elem;}

    // Mutator member functions
    void set_key(Key k) {itemKey = k;}
    void set_element(value_type e) {elem = e;}
private:
    Key itemKey;
    value_type elem;
};
```

Selection sort

- Consider a PriorityQueueSort method with the following declaration:

```
// Receives an array and orders it by priority
void PriorityQueueSort(value_type input[])
```

- The method:
  - takes an unsorted array of *n* elements
  - inserts the elements into a priority queue
  - repeatedly removes the minimum element from the priority queue, adding them to the original array, until the priority queue is empty
- The ordering of elements is done at the time of removal from the priority queue (i.e. when they are selected to be removed).
- Insertion of elements takes constant O(1) time (e.g. at the tail of the linked list). Selection of elements take O(n), i.e. the whole linked list has to be traversed to determine the one with the highest priority.

Insertion sort

- Consider a PriorityQueueSort method with the following declaration:

```
// Receives an array and orders it by priority
void PriorityQueueSort(value_type input[])
```

- The method:
  - takes an unsorted array of *n* elements
  - inserts the elements into a priority queue
  - repeatedly removes the minimum element from the priority queue, adding them to the original array, until the priority queue is empty

- The ordering of the elements is done at the time of insertion into the priority queue.
- Insertion of elements takes O(n) time, i.e. the linked list has to be traversed to determine the correct point of insertion to keep it ordered. Removal takes constant O(1) time (e.g. remove from the head of the linked list).

Complexity

- If you are trying to order n items using insertion sort, each insertion will take time O(n). Since you have n elements in total, the complexity then becomes $O(n*n) = O(n^2)$
- If you are trying to order n items using selection sort, each removal will take time O(n). Since you have n elements in total, the complexity is also $O(n^2)$
- This complexity is relatively high. A complexity of $O(n^2)$ indicates that the number of operations (comparisons, assignments, etc) required grows with the square of the size of the array.
- E.g. if sorting an array takes 10 seconds, sorting an array 2x as long will take 40 seconds. This level of computational complexity makes insertion sort and selection sort unsuitable for high performance applications.

Heap sort

- Heap sorting has a better performance compared to selection and insertion sorting, as instead of a linear data structure (linked list), it uses a binary tree.
- The heap used is a binary tree that:
  - stores item-key pairs at its internal nodes and null at its leaf nodes
  - satisfies the following properties:
    - for every internal node other than the root, the key stored at the node is less than or equal to the key stored at the node's parent
      - thus the element with the largest key will always be stored at the root node
    - is complete
      - all levels, except for the lowest level, have the maximum number of nodes
      - the lowest level is filled from the left to the right

Implementing sorting using a heap

- The priority queue *P* has instance data comprising:
  - a heap *T* for which each internal node *v* stores
    - an element of the priority queue
    - the key *k* for that element
  - a reference to the position of the last node of *T*
- The heap is itself implemented using one of the binary tree data structures presented previously
- The elements stored in the binary tree are key-item pairs

Enqueuing using a heap

- To implement the method `enqueue` of the priority queue ADT using *T*
  - a new internal node is added to *T*
    - this becomes the new last node of *T*
  - the new internal node is created by selecting the appropriate external node z and calling `expandExternal(z)` on it
  - This makes z an internal node with two `NULL` child nodes
  - z is usually the external node immediately to the right of the last insertion position
  - Exceptions to this occur when either
    - the current last node is the right-most node on its level, in which case z is the left-most node of the bottom level, or
    - there are no internal nodes in which case z is the root node of *T*

Enqueuing using a heap

  - after `expandExternal(z)` finishes
    - z becomes the new last node, and
    - the new data element-key pair (k, e) is stored there, i.e. $k(z) = k$
  - at this stage *T* is complete, but does not necessarily comply with the heap-order property
- unless z is the root of *T*, we need to compare *k(z)* with the key of z's parent *u*
  - if $k(u) < k(z)$ (in the case of "maximum root node") then heap-order has been violated by the insertion and the key-element pairs stored at *u* and z need to be swapped
  - this may violate heap-order higher up the tree, so swapping is continued until heap-order is not violated

Enqueuing using a heap

- The process of moving the new key-element pair into its correct position in the tree is called *up-heap bubbling*
  - in the worst case the bubbling causes the new key-element pair to move up to the root position
  - the cost of up-heap bubbling is proportional to the height of the tree, and since the tree height is given by h = log2(n + 1) the cost of up-heap bubbling is O(log n)
    - E.g. for an array of 1,023 elements, the associated heap will be a binary tree with 10 levels
- To start the process of heap insertion it is necessary to find the insertion position z
  - the heap keeps as instance data the last insertion position *w*

Enqueuing using a heap

- To find the insertion position using the binary tree ADT:
  - start at *w* and move up the tree calling the `parent()` function until either the root (determined by using `isRoot()`) or *w* is found to be a left-child (determined by using `isLeftChild`)
- The method `isLeftChild` would be implemented as follows:

```
bool isLeftChild(Position p)
{
    if (isRoot(p)) {return false;}
    else
        return (leftChild(parent(p)) == p);
}
```

Enqueuing using a heap

- If the root has been reached, then the last insertion position was the right-most internal node of its level
  - set *u* to the root
- If the parent of a left-child was reached
  - set *u* to be the sibling of the left-child (i.e. the right-child of the reached node's parent)
- Starting at *u*, move down the tree using `leftChild()` until an external node *z* is reached
  - this may involve zero or more calls to `leftChild()`
  - Insertion code follows on the next slide.

Enqueuing using a heap

- Insertion code

```
void insertItem(Key k, Item e)
{
        Position z; // Position to insert
        if (isEmpty())
                z = root();
        else
        {
                z = last;
                while (!isRoot(z) && !isLeftChild(z))
                        z = parent(z);
                if (!isRoot(z))
                        z = rightChild(parent(z));
                while (!isExternal(z))
                        z = leftChild(z);
        }
        expandExternal(z);
        replace(z, new Comp(k, e));
        last = z;
...
```

Enqueuing using a heap

- Insertion code (up heap bubbling)

```
// Up-heap bubbling comes next (max at root node)
…
Position u;

while (!isRoot(z)) //Up-heap bubble (z is the last
{                  //insertion position
    u = parent(z);
    if (z.get_comp().get_key() < u.get_comp().get_key()) {break;}
    swap(u, z);
    z = u;
}
```

Dequeuing using a heap

- The procedure for locating the new last node in an insertion can be reversed to update the last node after removal
- Removal from the heap is necessary to implement the dequeue method of the priority queue class
- It is known that the maximum element is stored at the root of the heap *T*
- unless the root is the only internal node we cannot simply delete the root node, because that would ruin the binary tree
- what we do is copy the key-element pair stored at the last node *w* into the root node
- then we delete the last node using `removeExternal(w)`

Dequeuing using a heap

- The procedure for locating the new last node in an insertion can be reversed to update the last node after removal
- Removal from the heap is necessary to implement the dequeue method of the priority queue class
- It is known that the maximum element is stored at the root of the heap *T*
- unless the root is the only internal node we cannot simply delete the root node, because that would ruin the binary tree
- what we do is copy the key-element pair stored at the last node *w* into the root node
- then we delete the last node using `removeExternal(w)`

Example

- Sorting a small vector

```
[ 5 1 8 4 10 3 6 9 ]
```

- Add all elements to a heap, and then remove them sequentially.

See you next week!