The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

PSets Heapsort, PSet Heap & PQ

Table of Contents

Heap

A heap is a tree-based data structure that satisfies the heap property: If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap.

A common implementation of a heap is the binary heap, in which the tree is a complete binary tree in an array structure.

In a maxheap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a minheap, the keys of parent nodes are always smaller than or equal to those of the children and smallest key is in the root node.

Maxheap vs. Minheap

As we know, the minheap has the exact same algorithm and data structure of the maxheap except the comparison the during sink or swim. In maxheap, we have used the function 'less()' consistently. Only difference between maxheap and minheap in the code is to use whether you use either less() or more(). Amazing, isn't it?

Instead of duplicating the functions after functions, we simply keep a function pointer that holds either less() or more() function depending on the current operation of the heap structure. In other words, instead of using less() or more() in the code everywhere, use the function pointer that points either less() or more().

For that purpose, the 'comp' function pointer can be defined in heap.h.

```
struct Heap {                       // heap or min/max priority queue
 int *nodes;                        // an array of nodes
 int capacity;                      // array size of node or key, item
 int N;                             // number of nodes in the heap
 bool(*comp)(heap, int, int);       // less() for max, more() for minheap
 Heap(int capa = 2) {
    capacity = capa;
    nodes = new int[capacity];
    N = 0;
    comp = nullptr;
 };
 ~Heap() {};
};
using heap = Heap *;
```

This function pointer is set to less( ) when the user selects maxheap, otherwise to more( ). Whenever you have used less( ), you replace it with the 'comp' function pointer. In other words, instead of using the hard-coded less( ) function you use the 'comp' function pointer. Then the CBT becomes maxheap or minheap depending on user's choice.

When the user chooses the option 'z', you may call the function **setType()** to set the compare function pointer (p->comp = ?).

```
// sets the compare function less() for maxheap, more() for minheap.
void setType(heap p, bool maxType) {
 p->comp = maxType ? ::less : more;    // less() uses global scope resolution ::
}
```

Priority Queue

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

The heap is one maximally efficient implementation of an abstract data type called a **priority queue**, and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented.

Heapsort

Heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a **heap data structure** rather than a linear-time search to find the maximum.

Heapsort algorithm

The heapsort algorithm involves preparing the list by first turning it into a **maxheap**. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values (N) considered in the heap operation by one, and sifting the new first value (root) into its position in the heap. This repeats until the range of considered values is one value in length.

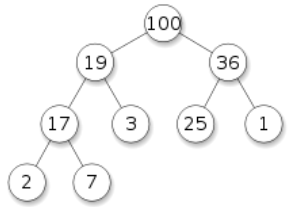
The steps are:

1. Build a maxheap/minheap (this process is called as **heapify**) from the input data. This process builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the sink() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

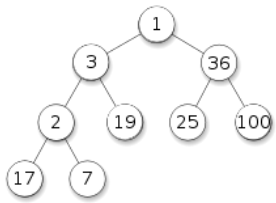
The heapify() operation is run once, and is O(n) in performance. The sink() function is O(log(n)), and is called n times. Therefore, the performance of this algorithm is O(n + n * log(n)) which evaluates to O(n log n).

Heapify

Heapify is the process of converting a complete binary tree into a Heap data structure. A heap must be a complete binary tree and satisfy the heap-order property, the value stored at each node is greater than or equal to its children.



The following is a maxheap data structure (root node contains the largest value). An array containing this Heap would look as {100, 19, 36, 17, 3, 25, 1, 2, 7}. Note the complete binary tree, left-justified and the heap-order where each parent is larger or equal to its children.



To arrive at the above heap structure, we might start with a binary tree that looks something like {1, 3, 36, 2, 19, 25, 100, 17, 7}

In heapify(), sink() will iterate across parent nodes comparing each with their children, **beginning at the last parent (the last internal node or 2 in this example) working backwards**, and swap them if the child is larger until we end up with the maxheap data structure.

Step 1.2: Tracing heap construction and heapsort

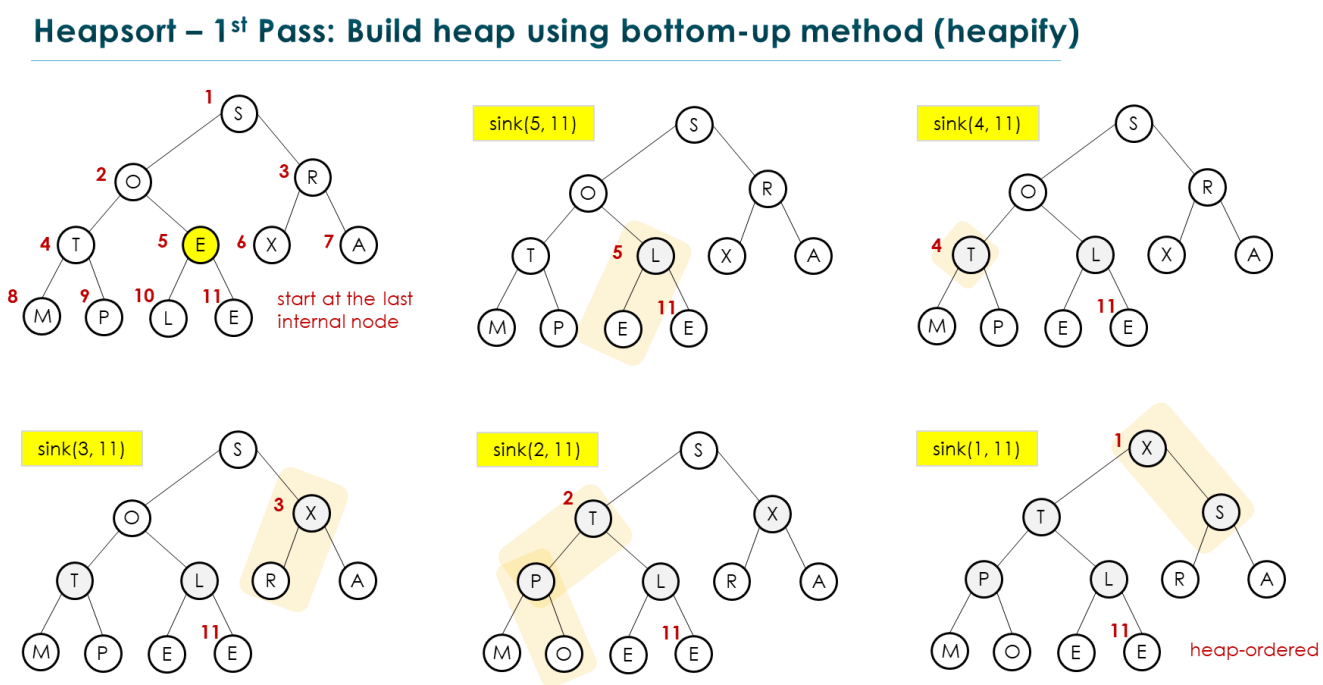
Let's understand the heap data structure and heapsort in this part of homework. It is important for you to understand the algorithm and how it work first. Heapsort algorithm consists of two steps. You may refer to the heapsort algorithm explained 'Warming-up' section above for detail.

Let us suppose that we have a small input list a[] shown below. Exclude the first element a[0] for simplicity.

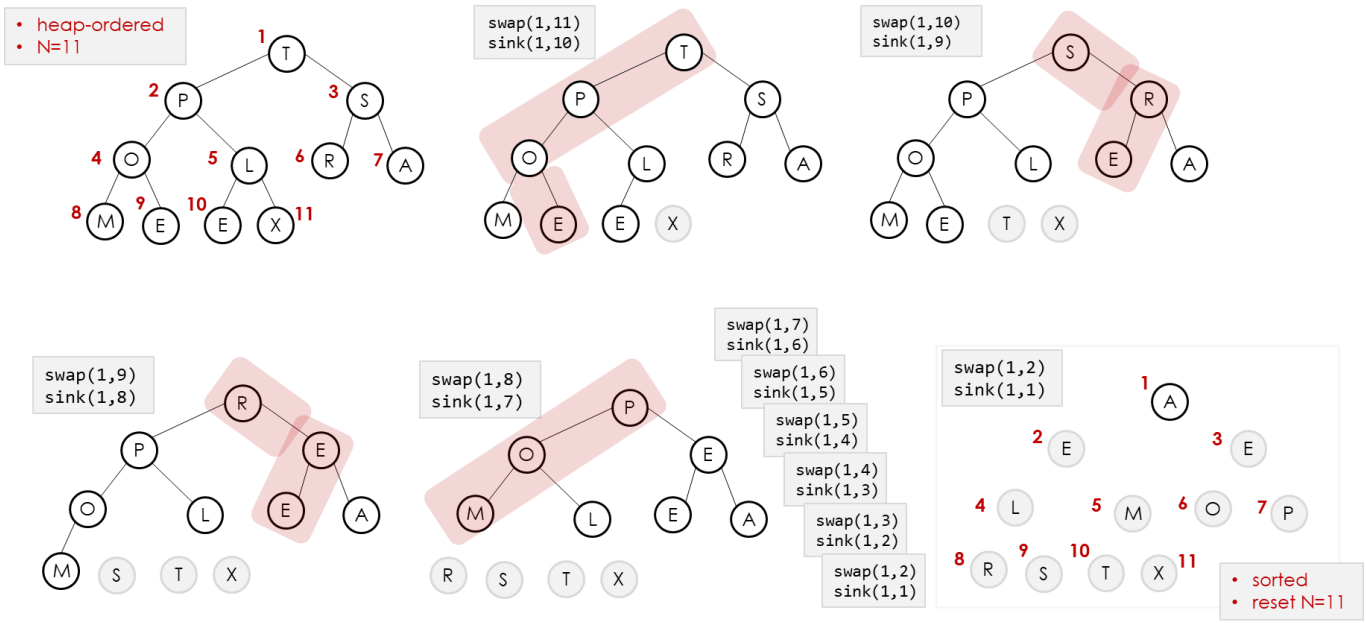
```
a[] = {' ', 'S', 'O', 'R', 'T', 'E', 'X', 'A', 'M', 'P', 'L', 'E'};
```

The first step makes the input array a[] into a maxheap. This process is called '**heapify**' that uses sink() repeatedly. The second step is getting the sorted output in place.

Let's draw every tree structure that changes during sorting. In each tree, write what values of N and k are and the sorted output status. In other words, turn in something similar to the following figure. Turn in a hand-drawn image file with a string given in the instruction.



Heapsort – 2nd Pass: Repeatedly remove the maximum key.



1. Your first job is to do a heapsort as shown above by hands on a piece of paper.
   - Use the word that consists of **your two initials at the end of "CHRISTIAN".**
     For example, my word to work with becomes "CHRISTIANYS" since my initials from the first name is YS. If your name is "Han, **D**ong**M**an", your word becomes "CHRISTIAN**DM**"
2. Questions:
   A. In 1st pass, list the first and the second characters you have to work with, and the number of comparisons to heapify them, respectively:
      1st pass _____, _____ comparisons _____, _____ comparisons
   B. In 2nd pass: In 2nd pass, list the first and the second characters to sink, and the number of comparisons during sinking them, respectively:
      2nd pass: _____, _____ comparisons _____, _____ comparisons

   For example, my case with "Y" and "S"
   A. 1st pass: "Y" _____ comparisons, "S" _____ comparisons
   B. 2nd pass: "Y" _____ comparisons, "S" _____ comparisons

- While following the two passes drawing patterns shown above, draw your heapsort process with your word ("CHRISTIAN_ _") where _ _ must be your initials.
- Include Questions and answers shown above in your submission.
- Make sure that your answer of this step matches with the result of your code in Step 1.2 or heapsortx.exe. Otherwise you may not get the credit for this step.

Step 1.2: Implement heap and heapsort in heapsort.cpp

Implement some helper functions such as swap(), less(), more(), sink(), swim() and heapify() first. Then you use those helper functions to implement heapsort(), grow() and trim() functions to test them. Make it work like heapsortx.exe provided for your jumpstart.

Once coded, test it with your word (CHRISITIAN_ _). **Follow the instructions,** if any, while coding your skeleton code file (heapsort.cpp).

Test it with a given input list.

- heapsort.cpp - All your work goes into this file.
- heapsortx.exe - Executable for testing, the 1st char is a blank.
- Build command line for heapsort:

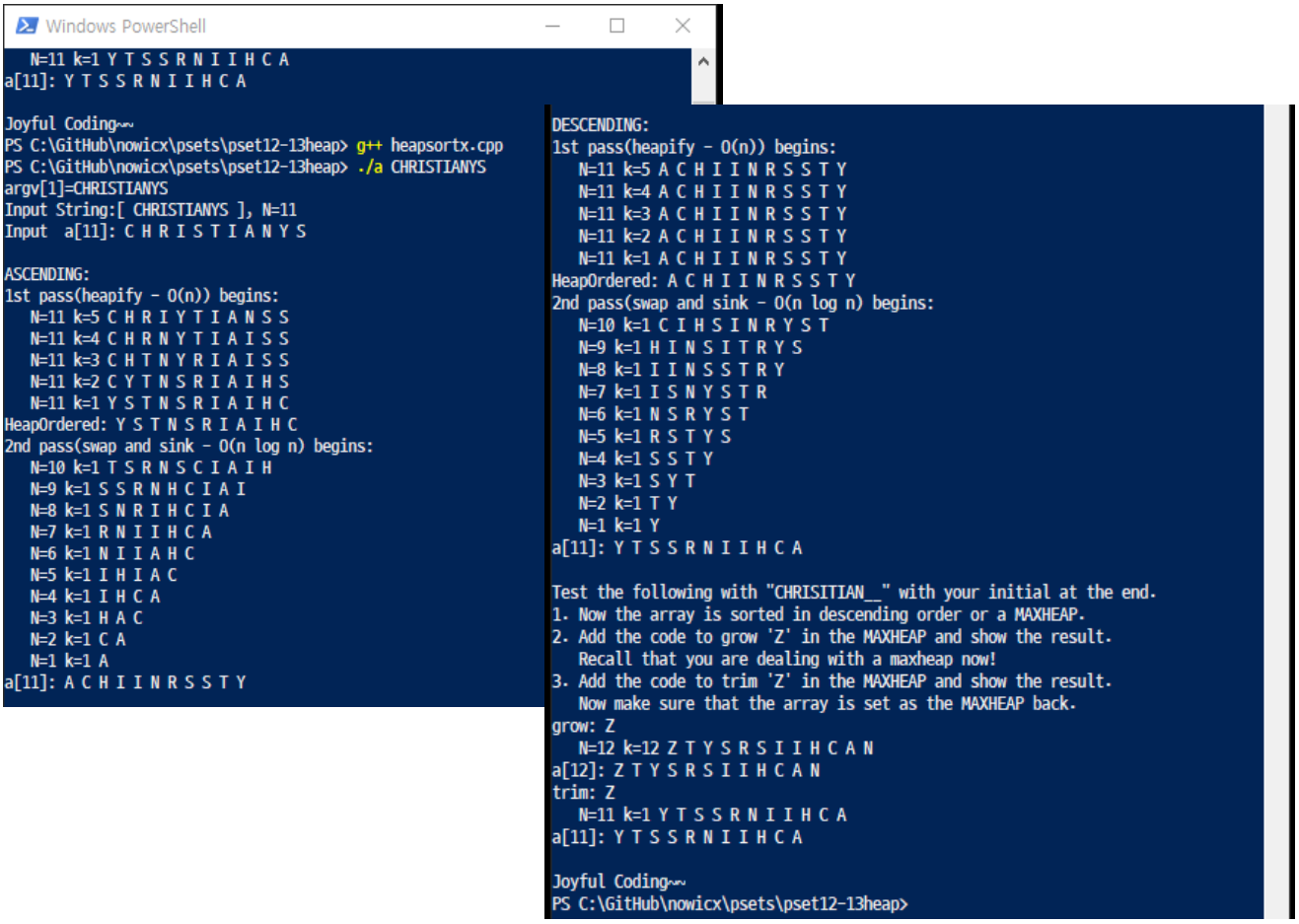
```
$ g++ heapsort.cpp -o heapsort                    #pc
$ g++ -std=c++11 heapsort.cpp -o heapsort         #mac
```

heapsort.cpp does not depend on any other external files or library. This is a good chance for you to understand the heap data structure and algorithms before you go for a full-blown heap & PQ implementation.

Hint: You may implement the **`heapsort()`** for ascending order first while using **`::less().`**

1. Once you make sure that it works for ascending order, then you replace where **less()** functions used in **`sink()`** and **`swim()`** with the **`comp`** function pointer.
   Notice that **`comp`** is defined as a global variable in this file for the simplicity.
2. In the **`main(),`** you set `comp` to **`less`** function for ascending order before invoking heapsort() and **`comp`** to **`more`** for descending order.
3. In general, you are not supposed to use global variables. In this example, however, a global variable, `comp`, is used for your convenience.
4. Test heapsort.cpp with your word ("CHRISTIAN_ _") where _ _ must be your initials.
5. Test heapsort.cpp with your word ("CHRISTIAN_ _") and grow("Z") and trim("Z").

Sample Run:



Complete Binary Tree(CBT) and Heap

As a warming up, you build a project called heap and display a complete binary tree. The following files are provided. In this step, you should **not include heapsort.cpp** that you used in Warming-up step. However, you still need to nowic.h and nowic.lib as usual.

- heap.h, tree.h - Don't change these files.
- heap.cpp - All of your work except heapprint() goes here.
- heapDriver.cpp - Change it only if it is necessary absolutely.
- heapprint.cpp - Implement heapprint() here.
- treeprint.cpp - Don't change these files.
- heapx.exe - Executable for the reference, DEBUG off version

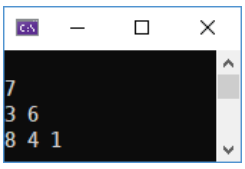
- Build command line. **Don't** include heapsort.cpp. For mac, add **–std=c++11**:

```
$ g++ heap.cpp heapDriver.cpp heapprint.cpp treeprint.cpp -I../include -L../lib
-lnowic -o heap
```

Step 2.1: growCBT(), trimCBT(), contains() and using reserve()

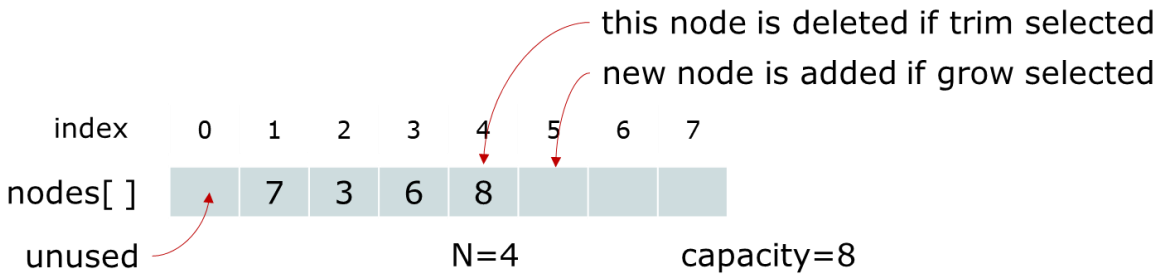
As you know, the complete binary tree is each level of the tree is filled, except possibly the bottom level. Even at the bottom level, it is filled from left to right. When you select a 'trim' menu, it will delete the last leaf node in the tree.

(1) Implement growCBT(), trimCBT() and contains() first.
When you choose 'g' or 't' menu in CBT menu, then it invokes either growCBT() or trimCBT() function, respectively. As you enter a new key, it should be inserted in the heap structure member called "nodes" array. It is inserted at the last position available of the tree. At "trim", it automatically deletes the last node in the tree.

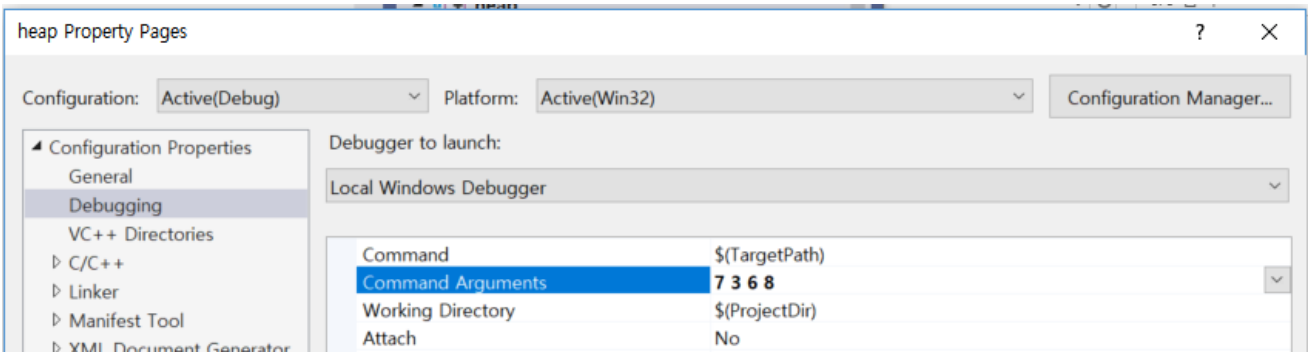


After you enters keys, for example 7 3 6 8 4 1 one by one, nodes[] in the heap are stored in memory as shown below:

When you have 7, 3, 6, and 8 in the heap, the heap has the following settings.



You may reach this status by initializing the command line arguments 7, 3, 6, and 8. In Visual Studio, go to the project properties → Configuration Properties → Debugging → Command Arguments:



(2) Use the **reserve()** function provided such that **growCBT()** and **trimCBT()** works even when it expands with many nodes or shrinks back.

Now, when you make the tree grow or insert more nodes in the tree, you may encounter problems since we created the tree with the exact size of the sample tree given at the beginning of the program. We would like to fix this problem first.

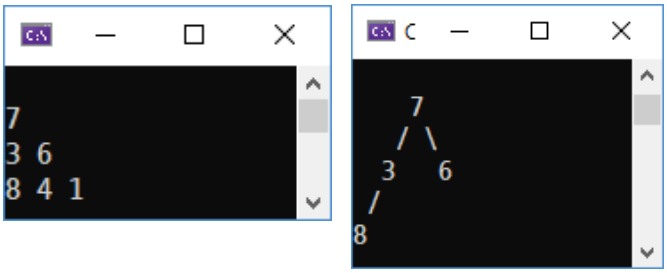
First, implement **reserve()** function that takes the heap and a new capacity (usually N * 2 or N/2) and reserves the node array in the heap.

Secondly, you must invoke the **reserve()** function to in **growCBT()**and **trimCBT()** to adjust the array size of node in the heap when necessary. You need to find when to do it. You may use the following principles.

You double the node array size when the size (the number of node, N) reaches the capacity (or the array allocation size). You reduce the array size (capacity) in half when the size (or N) reaches 1/4 of the capacity (or one quarter full).

Step 2.2: Implement heapprint() in heapprint.cpp

Instead of simply displaying a heap structure by level coded in heapprint_level() function, we would like to display it just like a tree.



For the display purpose, **we must construct a tree** from the heap data structure. Once we have a tree, we can use **treeprint()** functions available to display it.

There are two methods to construct a binary tree from a heap which is CBT I can think of:

Method I: Using queue

This algorithm uses a queue to build a binary tree(BT) from a complete binary tree(CBT) which is represented by an int array.

0. If the CBT size is zero, return a nullptr.
1. Create the tree (root) node (BT) with the first key from CBT (or nodes[1]).
2. Enqueue the tree root node.
3. Loop through **the CBT nodes[2] to nodes[N]**
   A. Make a new node from CBT nodes[].
   B. Get the tree node in the queue.
   C. If the left child of this tree node doesn't exist,
      set the new node to the left child of the tree node.
      else if the right child of this tree node doesn't exist,
      set as the new node to the right child
   D. If the tree node is already full, pop (or dequeue) it.
   E. Enqueue the new node (to add children later if any).
4. treeprint(root)

```
// Using queue, build binary tree from CBT
tree buildBT(heap p) {
  std::queue<tree> que;
  int N = size(p);
  tree root = new TreeNode{ p->nodes[1] };
  que.push(root);

  // your code here
}
```

Implement this buildBT() shown above in heapprint.cpp.

Method 2: Using recursion

This algorithm uses a recursion to build a binary tree(BT) from a complete binary tree which is represented by an int array. We have experienced to build an AVL tree from an array of int before. We may apply the similar algorithm here. Only difference between AVL and BT is the range of array to pass during the recursion. You may review the function buildAVL().

First of all, create a recursive function that creates a complete binary tree from an int array. This function takes an int array, starting index, and size of the array and returns the root as shown below:

```
tree buildBT(int *nodes, int i, int n) {
  If i > n, return nullptr – terminate condition
  Create the tree (root) node with nodes[i]).
    Invoke buildBT() for all its left children (or i * 2).
       Set its return to the left child of the root.
    Invoke buildBT() for all its right children (or i * 2 + 1).
       Set its return to the right child of the root.
  Return root
}
```

Implement this buildBT() shown above in heapprint.cpp.

How to test method 1 & method 2

Once you have buildBT()s, then, your job left is a piece of cake. For pedagogical purpose, we use the tree built by recursion if the tree size is an even number, otherwise use the tree built by iteration. Test it as shown below.

Don't forget invoking clear() after displaying the tree.

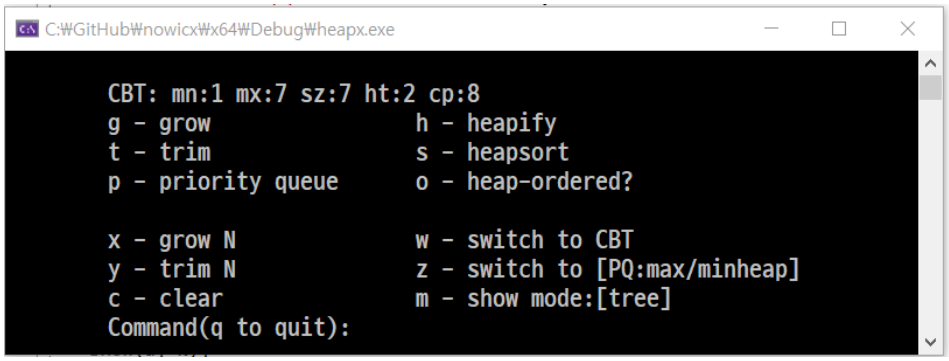
```
// print a heap using treeprint() - must build a tree to call treeprint()
void heapprint(heap p, int mode) {
 if (empty(p)) return;

  // build tree in two different ways for pedagogical purpose
 if (size(p) % 2 == 0) {
    cout << "\t[Tree built using recursion]\n";
    root = buildBT(p->nodes, 1, size(p));  // using recursion
 }
 else {
    cout << "\t[Tree built using iteration]\n";
    root = buildBT(p);                 // using iteration
 }

 switch (mode) {
 case TREE_MODE:
    treeprint(root);
    break;
 case LEVEL_MODE:
    treeprint_levelorder(root);
    break;
 default: // TASTY_MODE: show the fist and last few levels only
    treeprint_levelorder_tasty(root);
    cout << endl;
 }
  clear(root);
}
```

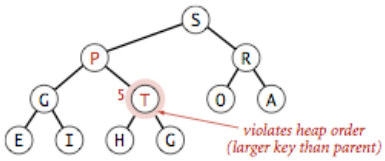
Step 2.3: Priority Queue(PQ): Maxheap/MinHeap

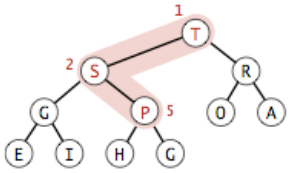
In this step, implement the heap abstract data types as given in the following menu.



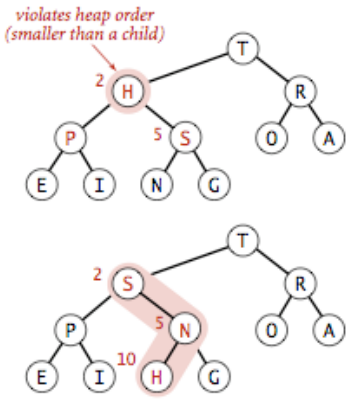
grow or insert: Implement the menu option **g**. We **add** the new item at **the end of the array** (or the last leaf node), increment the size of the heap, and then **swim up** through the heap with that item to restore the heap condition.

swim (Bottom-up reheapify): If the heap order is violated because a node's key becomes larger than that node's parents key, then we can make progress toward fixing the violation by exchanging the node with its parent. After the exchange, the node is larger than both its children (one is the old parent, and the other is smaller than the old parent because it was a child of that node) but the node may still be larger than its parent. We can fix that violation in the same way, and so forth, moving up the heap until we reach a node with a larger key, or the root.





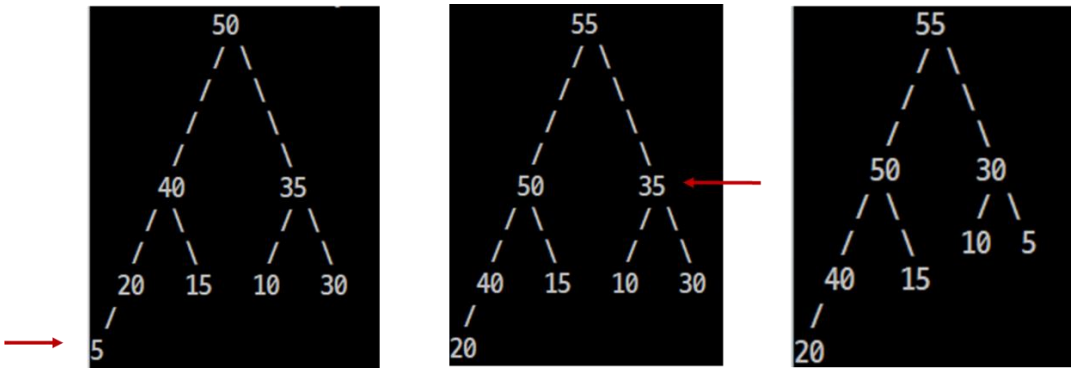
trim or delete: Implement the menu option t. We remove the maximum or the root in the maxheap. We take the largest item off the top, put the item from the end of the heap at the top, decrement the size of the heap, and then sink down through the heap with that item to restore the heap condition.



sink (Top-down heapify): If the heap order is violated because a node's key becomes smaller than one or both of that node's children's keys, then we can make progress toward fixing the violation by exchanging the node with the larger of its two children. This switch may cause a violation at the child; we fix that violation in the same way, and so forth, moving down the heap until we reach a node with both children smaller, or the bottom.

Priority queue (Replace): User may change one of the key values in the heap. The new node may go up or down depending on its value and the type of heap.

For example:
If you change 5 to 55, it will go up to the root and 20 is placed at the bottom.
If you change 35 to 5, 30 will go up where 35 is, then 5 goes down to the right corner.



Step 2.4: growN() & trimN()

It performs a user specified number of insertion or deletion of nodes in the heap.

- growN() inserts a user specified number N of nodes in the heap.
  - Find the max key in heap or CBT.
  - Allocate a key type array to store random keys.
  - Invoke randomN() to get random keys in the range [(max + 1)..(max + count)]
  - Invoke the function to insert keys in keys[], but one key at a time.
  - Optionally, print the heap if DEBUG is defined whenever a node is inserted.

- trimN() deletes a user specified number N of nodes in the heap.
  - If the number of node to delete is larger than the heap size, set the node count to the heap size.
  - Set a function pointer to the function to use for delete a node
  - Invoke the function to delete the node one by one.
  - Optionally, print the heap if DEBUG is defined whenever a node is deleted

Step 2.5: Improve or fix the algorithm

The heapOrdered() function checks the heap-ordered structure using recursion. In the following code (heap.cpp), there might be a room to improve it to run a little bit faster. Don't try to rewrite the entire code or introduce a new algorithm. For example, do not try to solve it using iteration.

Hint: stop the unnecessary computation or reduce unnecessary checking.

```
/** is it heap-ordered at a node k? */
bool heapOrderedAt(heap p, int k) {
  if (k > p->N) return true;

  int left  = 2 * k;
  int right = 2 * k + 1;

  if ((left <= p->N) && p->comp(p, k, left))  return false;
  if ((right <= p->N) && p->comp(p, k, right)) return false;

  return heapOrderedAt(p, left) && heapOrderedAt(p, right);
}

// is it heap-ordered?
bool heapOrdered(heap p) {
  if (empty(p)) return false;
```

```
 return heapOrderedAt(p, 1);
}
```

Submitting your solution

- Include the following line at the top of your every file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: _____
- Make sure your code **compiles** and **runs** right before you submit it. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the homework partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

Submit the following files for PSet Heapsort

- Step 1.1: Make one file including the followings.
  1. An image capture of your hand drawing
  2. An image capture of the heapsort code run. (Use heapsortx.exe or your own)
  3. Questions and your answers.
- Step 1.2: **heapsort.cpp**.

Submit the following file for PSet Heap & PQ(Priority Queue)

- **heapprint.cpp & heap.cpp**

Due and Grade points

- PSet 13: Heapsort
- PSet 14: Heap & PQ(Priority Queue)