Priority Queues and Huffman Trees

1. The Heap
   - Storing the heap with a vector
   - Deleting from the heap

2. Binary Search Trees
   - Sorting integer numbers
   - Deleting from a binary search tree

3. Huffman Trees
   - Encoding messages
   - A recursive tree creation algorithm
Priority Queues and Huffman Trees

1. **the Heap**
   - storing the heap with a vector
   - deleting from the heap

2. **Binary Search Trees**
   - sorting integer numbers
   - deleting from a binary search tree

3. **Huffman Trees**
   - encoding messages
   - a recursive tree creation algorithm
a heap as a vector

A heap is a binary tree:

For node at $p$: left child is at $2p + 1$, right child is at $2p + 2$. Parent of node at $p$ is at $(p - 1)/2$. 
Priority Queues and Huffman Trees

1. The Heap
   - storing the heap with a vector
   - deleting from the heap

2. Binary Search Trees
   - sorting integer numbers
   - deleting from a binary search tree

3. Huffman Trees
   - encoding messages
   - a recursive tree creation algorithm
popping from a heap

We replace the top first with the bottom, and then swap as long as parent less than largest child:
class Heap
{
    private:

    std::vector<int> h;

    int index_to_bottom;

    int max_child( int p );
    // returns index of largest
    // child or -1 if no child

    void swap_from_top( int p );
    // swaps the element at p with
    // its largest child if that child
    // has value larger than the parent
int Heap::max_child( int p )
{
    if(index_to_bottom <= p)
        return -1;
    else
    {
        int left = 2*p+1;
        int right = 2*p+2;
        if(left > index_to_bottom)
            return -1;
        else
        {
            if(right > index_to_bottom)
                return left;
            else
                return (h[left] > h[right]) ? left : right;
        }
    }
}
void Heap::pop() {
    if(index_to_bottom == 0)
        index_to_bottom--;  
    else {
        h[0] = h[index_to_bottom--];
        swap_from_top(0);
    }
}

void Heap::swap_from_top( int p ) {
    if(index_to_bottom == -1) return;
    int c = max_child(p);
    if(c == -1) return;
    if(h[c] > h[p]) {
        int t = h[p];
        h[p] = h[c]; h[c] = t;
        swap_from_top(c);
    }
}
sorting with a heap

Terminology: a heap is a *complete* binary tree.
A heap implements a *priority queue*.

```cpp
#include <queue>

using namespace std;

int main()
{
    priority_queue<int> q;
}
push(), top(), and pop()

Pushing \( n \) random numbers:

```c
for(int i=0; i<n; i++)
{
    int r = 10+rand() % 90;
    q.push(r);
}
```

Sorting with top and pop:

```c
vector<int> result;
for(; q.size() > 0; q.pop())
    result.push_back(q.top());
```

The numbers in \( \text{result} \) are in decreasing order.
Priority Queues and Huffman Trees

1. The Heap
   - Storing the heap with a vector
   - Deleting from the heap

2. Binary Search Trees
   - Sorting integer numbers
   - Deleting from a binary search tree

3. Huffman Trees
   - Encoding messages
   - A recursive tree creation algorithm
the Abstract Data Type of a binary search tree

abstract <typename T> binary_search_tree;
/* A binary search tree is a binary tree
   where everything less than the data element
   is in the left subtree and everything else
   is in the right subtree. */

abstract void insert ( binary_search_tree t, T item );
postcondition: if empty(t), then after
   insert(t, item) we have item = data_element(t),
it not empty(t), then:
   if item < data_element(t),
then item is in left_subtree(t),
   if item >= data_element(t),
then item is in right_subtree(t);
Sorting Numbers using a Tree

Consider the sequence 4, 5, 2, 3, 8, 1, 7

Insert the numbers in a tree:

```
   4
  /\  \
 2 5
/  \  /
1 3 8 \/
   7
```

Rules to insert $x$ at node $N$:
- if $N$ is empty, then put $x$ in $N$
- if $x < N$, insert $x$ to the left of $N$
- if $x \geq N$, insert $x$ to the right of $N$

Recursive printing: left, node, right sorts the sequence.
finding smallest element

Recursive algorithm to find smallest element?

Tree Tree::smallest() const
{
    if(root == NULL)
        return NULL;
    else if(root->left == NULL)
        return Tree(root);  
    else
    {
        Tree L = this->get_left();
        return L.smallest();
    }
}
Priority Queues and Huffman Trees

1. **the Heap**
   - storing the heap with a vector
   - deleting from the heap

2. **Binary Search Trees**
   - sorting integer numbers
   - deleting from a binary search tree

3. **Huffman Trees**
   - encoding messages
   - a recursive tree creation algorithm
deleting the root

Assume the binary search tree is not empty. We want to delete the root node.

- right or left child is null:

```
  r  \
   \  
    S  
```

→ removing root $r$ leads to subtree $S$

- left or right child is null:

```
  2  \
   \  
    1 3
```

$\rightarrow$

```
  3  \
   \  
    1 4
```

- otherwise ...
otherwise ... the general case

Removing the root:

![Diagram of a binary tree before and after removing the root]

**Algorithm:**

1. Find the smallest element of the right child.
2. Replace the root with that smallest element.
3. Update the parent of node of smallest element.

Observe that the smallest element has no left child.
finding the parent node

We have already code to find the smallest element. Needed: find the parent of the smallest node.

The base cases:

```cpp
Node* Tree::find_parent_node(int item) const
{
    if(root == NULL)
        return NULL;
    else if(root->data == item)
        return NULL;
    else
    {
        // general case on the next slide
    }
}
bool found = false;
Node *parent = root;
Node *child;
do {
    if(item < parent->data)
        child = parent->left;
    else
        child = parent->right;
    if(child == NULL)
        return NULL;
    else if(child->data == item)
        found = true;
    else
        parent = child;
} while(!found);
return parent;
removing any item

Knowing the removal of the root of a binary search tree, can we work with a “local root”?

Given an item that occurs in the search tree:

1. Find the item and its parent.
2. Consider the item as a “local root” node.
3. Update the appropriate child of the parent with the tree that has the item removed.

Why does this approach work?
Priority Queues and Huffman Trees

1. The Heap
   - storing the heap with a vector
   - deleting from the heap

2. Binary Search Trees
   - sorting integer numbers
   - deleting from a binary search tree

3. Huffman Trees
   - encoding messages
   - a recursive tree creation algorithm
binary codes

A Huffman tree is a binary tree with data at leaves. Turn left: add 0, turn right: add 1 to code.

Vowels e, a, o are more frequent than o and u.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>binary code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>o</td>
<td>1</td>
<td>0</td>
<td>1 0</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

Decode bit string by walking the tree, encode: use table.
encoding messages

A is the alphabet \{a_1, a_2, \ldots, a_n\} of symbols, a message \(M\) is a sequence of elements of \(A\).

\(f_M(a_i)\) is the frequency of the symbol \(a_i\) occurring in \(M\)

\(d_H(a_i) = \) depth of \(a_i\) in Huffman tree

\(= \) #bits in encoding of \(a_i\)

#bits of \(M\) as encoded by Huffman tree \(H\):

\[|M|_H = \sum_{a \in A} f_M(a) \times d_H(a)\]

Goal: find optimal Huffman tree \(H\) for message \(M\).
Priority Queues and Huffman Trees

1. The Heap
   - Storing the heap with a vector
   - Deleting from the heap

2. Binary Search Trees
   - Sorting integer numbers
   - Deleting from a binary search tree

3. Huffman Trees
   - Encoding messages
   - A recursive tree creation algorithm
a recursive idea

For two symbols, \( n = 2 \): \( a_1 \mapsto 0, a_2 \mapsto 1 \).

Order symbols along their frequencies in message \( M \):

\[
f_M(a_1) \leq f_M(a_2) \leq f_M(a_3) \leq \cdots \leq f_M(a_n)
\]

Replace \( a_1 \) and \( a_2 \) by \( a_{1,2} \), \( f_M(a_{1,2}) = f_M(a_1) + f_M(a_2) \),
then \( M_{n-1} \) is the message \( M \) over \( \{a_{1,2}, a_3, \ldots, a_n\} \)
and let \( H_{n-1} \) be the optimal Huffman tree to encode \( M_{n-1} \).

A Huffman tree \( H \) for \( M \) is then obtained via

Claim: this \( H \) obtained recursively is optimal for \( M \).
running the algorithm

\[
\begin{array}{c|c}
\cdot & f_5(\cdot) \\
\hline
e & 103 \\
a & 64 \\
o & 63 \\
i & 57 \\
u & 23 \\
\end{array}
\quad\quad
\begin{array}{c|c}
\cdot & f_4(\cdot) \\
\hline
e & 103 \\
iu & 80 \\
a & 64 \\
o & 63 \\
\end{array}
\quad\quad
\begin{array}{c|c}
\cdot & f_3(\cdot) \\
\hline
ao & 127 \\
e & 103 \\
iu & 80 \\
\end{array}
\quad\quad
\begin{array}{c|c}
\cdot & f_2(\cdot) \\
\hline
eiu & 183 \\
au & 127 \\
\end{array}
\]

Introduction to Data Structures (MCS 360)  Priority Queues and Huffman Trees  L-26  27 October 2017  27 / 32
running the algorithm

\[
\begin{array}{c|c}
\cdot & f_5(\cdot) \\
\hline
e & 103 \\
a & 64 \\
o & 63 \\
i & 57 \\
u & 23 \\
\end{array}
\quad \rightarrow 
\begin{array}{c|c}
\cdot & f_4(\cdot) \\
\hline
e & 103 \\
iu & 80 \\
a & 64 \\
o & 63 \\
\end{array}
\quad \rightarrow 
\begin{array}{c|c}
\cdot & f_3(\cdot) \\
\hline
ao & 127 \\
e & 103 \\
iu & 80 \\
\end{array}
\]
running the algorithm

<table>
<thead>
<tr>
<th></th>
<th>$f_5(\cdot)$</th>
<th></th>
<th>$f_4(\cdot)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>103</td>
<td>e</td>
<td>103</td>
</tr>
<tr>
<td>a</td>
<td>64</td>
<td>iu</td>
<td>80</td>
</tr>
<tr>
<td>o</td>
<td>63</td>
<td>a</td>
<td>64</td>
</tr>
<tr>
<td>i</td>
<td>57</td>
<td>o</td>
<td>63</td>
</tr>
<tr>
<td>u</td>
<td>23</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$

```
  0          1
 /   \      /   \      /   \      /   \
0       1   0       1   0       1
a       o   e       iu
```
running the algorithm

<table>
<thead>
<tr>
<th>.</th>
<th>( f_5(\cdot) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>103</td>
</tr>
<tr>
<td>a</td>
<td>64</td>
</tr>
<tr>
<td>o</td>
<td>63</td>
</tr>
<tr>
<td>i</td>
<td>57</td>
</tr>
<tr>
<td>u</td>
<td>23</td>
</tr>
</tbody>
</table>

![Huffman Tree Diagram](image)
Notes on Huffman Trees

A Huffman tree is a *full* binary tree: every node has two nonempty children.

Optimal Huffman trees are not unique.

Elements of an algorithm for Huffman Trees:

- Make frequency table of symbols in a text. Nodes in priority queue are of type `struct` containing `string` for the symbol and `int` for count.

- Recursive tree creation algorithm. Going forward: contract the frequency table. Returns from the recursive calls refines the tree.

Symbols occurring with least frequency will appear separately only in the last stage of the algorithm.
Summary + Exercises

Ended chapter 8 on trees.

Exercises:

1. Write an iterative version of the `smallest()` method to return the smallest element in a binary search tree.

2. Give code to use a STL priority queue to create a frequency table for lower case letters in a string.

3. Define a binary search tree for strings and use it to store words that appear in a text on file. A word is separated by one or more spaces.

4. Define the algorithm to decode a message (given as bit string) using a Huffman tree.