Applications of Maps

1. Performance of Hash Tables
   - chaining with vector of pairs
   - on the expected number of probes

2. Huffman Codes
   - minimal bit encoding
   - using the Huffman tree
   - building the Huffman tree
Applications of Maps

1. Performance of Hash Tables
   - chaining with vector of pairs
   - on the expected number of probes

2. Huffman Codes
   - minimal bit encoding
   - using the Huffman tree
   - building the Huffman tree
Chaining or bucket hashing works with linked lists:

An alternative implementation uses a vector of pairs:

For list of strings: `vector<pair<string, int>>` the `int` is an index pointer to next element.
Applications of Maps

1. Performance of Hash Tables
   - chaining with vector of pairs
   - on the expected number of probes

2. Huffman Codes
   - minimal bit encoding
   - using the Huffman tree
   - building the Huffman tree
expected number of probes

Denote \( K \) = number of keys in the table
\[ n = \text{size of the table} \]
then \( L = K/n \) is the load factor, \( L \in [0, 1] \).

Assuming the hash function gives an uniform distribution of the keys in the table, \( L \) is the average bucket size.

Then the expected number of probes is \( c = 1 + \frac{L}{2} \).

For open addressing: \( c = \frac{1}{2} \left( 1 + \frac{1}{1 - L} \right) \).

(See Donald E. Knuth: *Sorting and Searching*, volume 3 of the Art of Computer Programming for a proof.)
For load factor $L \in [0, 1]$, we compare $c = 1 + \frac{L}{2}$ (chaining) with $c = \frac{1}{2} \left( 1 + \frac{1}{1 - L} \right)$ (open addressing):

Open addressing is no longer competitive for $L > 0.6$. 
Hash Tables versus Search Trees

Concerns about hash tables:

1. An unsuccessful search in a hash table is wasted. If an item is not yet accounted for, then we often want to insert it to our collection and binary searches give either smaller or larger key after unsuccessful search.

2. Predicting the allocation for hash tables is difficult.

3. Good hashing methods work well on average but perform often terrible in the worst case.

However: binary search requires $O(\log_2(n))$ comparisons (e.g.: $n = 1,024$ requires 10 comparisons). With bucket hashing, we always get less than 2 probes. Keeping items in bucket sorted improves performance.
Applications of Maps

1. Performance of Hash Tables
   - chaining with vector of pairs
   - on the expected number of probes

2. Huffman Codes
   - minimal bit encoding
   - using the Huffman tree
   - building the Huffman tree
Encoding "abracadabra" with minimal #bits:

- a: 0
- b: 110
- c: 1110
- d: 1111
- r: 10

The binary code is 01101001110011110110100 and uses a minimal number of bits.
running the example

the Huffman tree:
arbcd
    a
   rbc
    r
   bcd
      b
     cd
       c
d

the Huffman code:
<a,0><b,110><c,1110><d,1111><r,10>
"abracadabra" is 01101001110011110110100
counting characters

give a message: abracadabra
the frequency table as map:
<a,5><b,2><c,1><d,1><r,2>
the frequency table:
<c,1><d,1><b,2><r,2><a,5>

A map is a natural data structure to compute a frequency table for characters in a string.

For the algorithm, we contract the least frequently occurring characters, so we need to sort the frequencies.

The elements of the set are of type `pair<int, char>` sorted on the frequency.
frequency table

map<char, int> frequency_table(string s)
{
    map<char, int> M;

    for(int i=0; i<s.size(); i++)
    {
        char c = s[i];
        if(M.find(c) == M.end())
            M[c] = 1;
        else
            M[c]++;
    }
    return M;
}
converting maps into sets

```cpp
set< pair<int,string> > convert( map<char,int> M )
{
    set< pair<int,string> > S;

    for(map<char,int>::const_iterator
        i = M.begin(); i != M.end(); i++)
    {
        pair<int,string> p;
        p.first = i->second;
        p.second = i->first;
        S.insert(p);
    }

    return S;
}
```
Applications of Maps

1. Performance of Hash Tables
   - chaining with vector of pairs
   - on the expected number of probes

2. Huffman Codes
   - minimal bit encoding
   - using the Huffman tree
   - building the Huffman tree
the Huffman tree

Navigating the Huffman tree:
  • turn left: write 0,
  • turn right: write 1.

At leaf we have a string of bits:
the code for the character at the leaf.

We recycle the binary tree for arithmetical expressions.

Data at the nodes are strings:
  • at leaf: store character in message
  • at node: keep contractions of characters
from tree to code

```c
void Huffman_Code
    ( Tree h, map<char,string>& c, string b )
{
    if(h.is_left_null() && h.is_right_null())
        c[h.get_data()[0]] = b;
    else
    {
        if(!h.is_left_null())
            Huffman_Code(h.get_left(),c,b+'0');
        if(!h.is_right_null())
            Huffman_Code(h.get_right(),c,b+'1');
    }
}
```
encoding character string

string encode( string s, map<char,string> c )
{
    string r = "";
    for(int i=0; i<s.size(); i++)
        r = r + c[s[i]];
    return r;
}
string decode( string s, Tree h )
{
    string r = "";
    Tree w = h;

    for(int i=0; i<s.size(); i++)
    {
        if(s[i] == '0') w = w.get_left();
        if(s[i] == '1') w = w.get_right();
        if(w.is_left_null() && w.is_right_null())
        {
            r = r + w.get_data();
            w = h;
        }
    }
    return r;
}
Applications of Maps

1. Performance of Hash Tables
   - chaining with vector of pairs
   - on the expected number of probes

2. Huffman Codes
   - minimal bit encoding
   - using the Huffman tree
   - building the Huffman tree
running the algorithm

abracadabra

Contracting the least frequently occurring characters, we reduce towards the base case:

\[
\begin{array}{c|c}
\cdot & f_5(\cdot) \\
\hline
c & 1 \\
d & 1 \\
b & 2 \\
r & 2 \\
a & 5 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c}
\cdot & f_4(\cdot) \\
\hline
b & 2 \\
\text{cd} & 2 \\
r & 2 \\
a & 5 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c}
\cdot & f_3(\cdot) \\
\hline
r & 2 \\
\text{bcd} & 4 \\
a & 5 \\
r\text{bcd} & \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c}
\cdot & f_2(\cdot) \\
\hline
a & 5 \\
r\text{bcd} & 6 \\
\end{array}
\]
splitting a leaf

Returning from the recursive call, we split the leaf we contracted before the call.

\[
\begin{array}{c|c}
    c & 1 \\
    d & 1 \\
    b & 2 \\
    r & 2 \\
    a & 5 \\
\end{array}
\]

\[
\begin{array}{c|c}
    f_5(\cdot) & \cdot \\
\end{array}
\]

\[
\begin{array}{c|c}
    f_4(\cdot) & \cdot \\
    b & 2 \\
    cd & 2 \\
    r & 2 \\
    a & 5 \\
\end{array}
\]

\[
\begin{array}{c|c}
    f_3(\cdot) & \cdot \\
    r & 2 \\
    bcd & 4 \\
    a & 5 \\
\end{array}
\]

\[
\begin{array}{c|c}
    f_2(\cdot) & \cdot \\
    a & 5 \\
    r bcd & 6 \\
\end{array}
\]
splitting a leaf

Returning from the recursive call, we split the leaf we contracted *before* the call.
a recursive algorithm

Building a tree from a frequency table:

<table>
<thead>
<tr>
<th>f_5(\cdot)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
</tbody>
</table>

Given the frequency table as ordered set:

1. reduce to base case making new set,
2. in the base case return tree with 2 leaves,
3. after the recursive call:
   1. search in the returned tree for the leaf,
   2. split the leaf along the contraction.
the base case

Tree Huffman_Tree ( set< pair<int,string> > S )
{
    set< pair<int,string> >::const_iterator i;
    i = S.begin();

    pair<int,string> p = *(i++);
    pair<int,string> q = *(i++);

    if(S.size() == 2)
    {
        string pq = p.second + q.second;

        Tree T(pq, Tree(p.second), Tree(q.second));

        return T;
    }
}
else
{
    pair<int,string> r;
    r.first = p.first + q.first;
    r.second = p.second + q.second;

    set< pair<int,string> > U;
    U.insert(r);
    while(i != S.end()) U.insert(*(i++));

    Tree T = Huffman_Tree(U);
    pair<Tree,bool> Z;
    Z = split(T,r.second,p.second,q.second);

    return Z.first;
}
splitting leaves

split returns Tree and boolean for result and to indicate if string was found.

```cpp
pair<Tree,bool> split
    ( Tree T, string r, string p, string q )
{
    pair<Tree,bool> result;

    if(T.get_data() == r)
    {
        cout << "split " << r << endl;
        result.first = Tree(r,Tree(p),Tree(q));
        result.second = true;
    }
```
else
{
    result.second = false;
    if(!T.is_right_null())
    {
        pair<Tree,bool> R;
        R = split(T.get_right(),r,p,q);

        if(R.second)
        {
            result.first = Tree(T.get_data(),
                                  T.get_left(),R.first);
            result.second = true;
        }
    }
}
if(!result.second)
{
    if(!T.is_left_null())
    {
        pair<Tree,bool> L;
        L = split(T.get_left(),r,p,q);

        if(L.second)
        {
            result.first = Tree(T.get_data(),
                                L.first,T.get_right());
            result.second = true;
        }
    }
}
the main program

```cpp
int main()
{
    cout << "give a message : ";
    string message;
    getline(cin,message,'\n');

    map<char,int> M = frequency_table(message);
    cout << "the frequency table as map :" << endl;
    write(M); cout << endl;

    set< pair<int,string> > S = convert(M);
    cout << "the frequency table :");
    write(S); cout << endl;
}
```
Tree \( T = \text{Huffman\_Tree}(S) \);
cout << "the Huffman tree :" << endl;
write\_with\_depth(0,T);

map<char,string> c;
Huffman\_Code(T,c,"");
cout << "the Huffman code : ";
write(c); cout << endl;

string coded\_message = encode(message,c);
cout << "encoding of \"" << message << \"\" is " << coded\_message << endl;
string decoded\_message = decode(coded\_message,T);
cout << "decoding of \"" << coded\_message << \"\" is " << decoded\_message << endl;
Popular Dehong City Hui Autonomous Prefecture High School

Introduction to Data Structures (MCS 360)

Applications of Maps

L-29 3 November 2017 31 / 31

Summary + Exercises

Ended chapter 9.

Exercises:

1. Describe the adjustments to the Hash_Table in the mcs360_chain_hash_table.h of last lecture to work with a vector of pairs (data and index pointer to the next element in the list) instead of an STL list.

2. Run the algorithm to create a Huffman tree by hand on the string "abcd". Draw all intermediate trees.