Heaps, Priority Queues, Compression

- Compression is a high-profile application
  - .zip, .mp3, .jpg, .gif, .gz, ...
  - Why is compression important?

- What's the difference between compression for .mp3 files and compression for .zip files? Between .gif and .jpg?
  - What's the source, what's the destination?
  - Why does the difference make a difference?

- Is it possible to compress (lossless compression rather than lossy) every file? Every file of a given size?
  - What are repercussions?

Priority Queue

- Compression motivates the study of the ADT priority queue
  - Supports two basic operations
    - `insert` — an element into the priority queue
    - `delete` — the minimal element from the priority queue
  - Implementations may allow `getmin` separate from `delete`
    - Analogous to `top/pop`, `front/dequeue` in stacks, queues
  - Simple sorting using priority queue (see `pqdemo.cpp` and `usepq.cpp`)

```cpp
string s; priority_queue pq;
while (cin >> s) pq.insert(s);
while (pq.size() > 0) {
    pq.deletemin(s);
    cout << s << endl;
}
```

Priority Queue implementations

- Implementing priority queues: average and worst case

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Getmin</th>
<th>DeleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted vector</td>
<td>O(...)</td>
<td>O(...)</td>
<td>O(...)</td>
</tr>
<tr>
<td>Sorted vector</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list (sorted?)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Balanced tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quick look at class `tpq<...>`

- Templated class like `tstack`, `tqueue`, `tvector`, `tmap`, ...
  - If `deletemin` is supported, what properties must types put into `tpq` have, e.g., can we insert `string`? `double`? `struct`?
  - Can we change what minimal means (think about `anaword` and `sorting`)?

- If we use a `compare` function object for comparing entries we can make a `min-heap` act like a `max-heap`, see `pqdemo.cpp`
  - Notice that `RevComp` inherits from `Comparer<Kind>`
  - How is `Comparer` accessed?

- How is this as a sorting method, consider a vector of elements.
  - In practice `heapsort` uses the vector as the priority queue
  - From a big-Oh perspective no difference: $O(n \log n)$
    - Is there a difference? What's hidden with $O$ notation?
Priority Queue implementation

- The class in tpq.h uses heaps, very fast and reasonably simple
  ▶ Why not use inheritance hierarchy as was used with tmap?
  ▶ Trade-offs when using HMap and BSTMap:
    • Time, space
    • Ordering properties

- Mechanism for changing comparisons used for priority
  ▶ Different from comparison used in sortall functions (anaword)
    • Functions are different from classes when templates used
    • Functions instantiated when called, object/class instantiated when object constructed
  ▶ The tpq mechanism uses inheritance, sorting doesn’t
    • In theory we could have template function in non-templated class, but g++ doesn’t support template member functions

Creating Heaps

- Heap is an array-based implementation of a binary tree used for implementing priority queues, supports:
  ▶ insert, findmin, deletemin: complexities?

- Using array minimizes storage (no explicit pointers), faster too --- children are located by index/position in array

- Heap is a binary tree with shape property, heap/value property
  ▶ shape: tree filled at all levels (except perhaps last) and filled left-to-right (complete binary tree)
  ▶ each node has value smaller than both children

Array-based heap

- store “node values” in array beginning at index 1
- for node with index k
  ▶ left child: index 2*k
  ▶ right child: index 2*k+1

- why is this conducive for maintaining heap shape?
- what about heap property?
- is the heap a search tree?
- where is minimal node?
- where are nodes added? deleted?

Adding values to heap

- to maintain heap shape, must add new value in left-to-right order of last level
  ▶ could violate heap property
  ▶ move value “up” if too small

- change places with parent if heap property violated
  ▶ stop when parent is smaller
  ▶ stop when root is reached

- pull parent down, swapping isn’t necessary (optimization)
Adding values, details

```cpp
void pqqueue::insert(int elt) {
  // add elt to heap in myList
  myList.push_back(elt);
  int loc = myList.size();
  while (1 < loc &&
         elt < myList[loc/2]) {
    myList[loc] = myList[loc/2];
    loc /= 2;  // go to parent
  }
  // what's true here?
  myList[loc] = elt;
}
```

Removing minimal element

- Where is minimal element?
  - If we remove it, what changes, shape/property?
- How can we maintain shape?
  - "last" element moves to root
  - What property is violated?
- After moving last element, subtrees of root are heaps, why?
  - Move root down (pull child up) does it matter where?
- When can we stop "re-heaping"?

Huffman codes and compression

- Compression exploits redundancy
  - Run-length encoding: 0001110010000
    - Coded as 3421113
    - Useful? Problems?
  - What about 1010101010101010101?

- Encoding can be based on characters, chunks, ...
  - Instead of using 8-bits for 'A', use 2-bits and 14 bits for 'Z'
    - Why might this be advantageous?
  - Methods can exploit local information
    - abcabcabc is 3(abc)
  - Huffman coding is optimal per-character coding method

Towards Compression

- Each ASCII character is represented by 8 bits, one byte
  - bit is a binary digit, byte is a binary term
  - compress text: use fewer bits for frequent characters (does this come free?)
- 256 character values, $2^8 = 256$, how many bits needed for 7 characters? for 38 characters? for 125 characters?

ASCII: 13 x 8 = 104 bits
3 bit code: 13 x 3 = 39 bits
compressed: ???

<table>
<thead>
<tr>
<th>go go gophers:</th>
<th>8 different characters</th>
<th>ASCII</th>
<th>3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103 1100111 000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>111 1101211 001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>112 1110000 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>104 1101000 011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>101 1100101 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>114 1110010 101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>115 1110011 110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp.</td>
<td>32 1000000 111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Huffman coding: go go gophers

#### ASCII 3 bits Huffman

- **g**: 103 1100111 000 10
- **o**: 111 1101122 001 2
- **p**: 112 1110000 010 3
- **h**: 104 1101000 011 3
- **e**: 101 1100101 100 1
- **r**: 114 1110010 101 2
- **s**: 115 1110011 110 2
- **sp.**: 32 1000000 111 2

- Choose two smallest weights
  - Combine nodes + weights
  - Repeat
  - Priority queue?

- Encoding uses tree:
  - 0 left/1 right
  - How many bits?

#### ASCII and Huffman Codes

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>10</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>2</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>2</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>2</td>
</tr>
</tbody>
</table>

### Properties of Huffman code

- Prefix property, no code is prefix of another code
- Optimal per character compression
- Where do frequencies come from?
- Decode: need tree

#### Example Code: 1001111010011100001101111110001

### Trie: efficient search of words/suffixes

- A trie (from retrieval, but pronounced “try”) supports
  - These operations are $O(\text{size of string})$ regardless of how many strings are stored in the trie!
    - Insert/Delete string
    - Lookup string or string prefix

- In some ways a trie is like a 128 (or 26 or alphabet-size) tree, one branch/edge for each character/letter
  - Node stores branches to other nodes
  - Node stores whether it ends the string from root to it

- Extremely useful in DNA/string processing
  - Monkeys and typewriter simulation: similar to statistical methods used in Natural Language understanding

### Trie picture and code (see trie.cpp)

- To add string
  - Start at root, for each char create node as needed, go down tree, mark last node

- To find string
  - Start at root, follow links
  - If Null/0 not contained
  - Check word flag in node

- To print all nodes
  - Visit every node, build string as nodes traversed

- What about union and intersection?

- Indicates word ends here
Sorting: From Theory to Practice

- Why do we study sorting?
  - Because we have to
  - Because sorting is beautiful
  - Because ...

- There are \( n \) sorting algorithms, how many should we study?
  - \( O(n) \), \( O(\log n) \), ...
  - Why do we study more than one algorithm?
    - ...
  - Which sorting algorithm is best?

Sorting out sorts (see also sortall.cpp)

- Simple, \( O(n^2) \) sorts --- for sorting \( n \) elements
  - Selection sort --- \( n^2 \) comparisons, \( n \) swaps, easy to code
  - Insertion sort --- \( n^2 \) comparisons, \( n^2 \) moves, stable, fast
  - Bubble sort --- \( n^2 \) everything, slow, slower, and ugly

- Divide and conquer faster sorts: \( O(n \log n) \) for \( n \) elements
  - Quick sort: fast in practice, \( O(n^2) \) worst case
  - Merge sort: good worst case, great for linked lists, uses extra storage for vectors/arrays

- Other sorts:
  - Heap sort, basically priority queue sorting
  - Radix sort: doesn’t compare keys, uses digits/characters
  - Shell sort: quasi-insertion, fast in practice, non-recursive

Selection sort

- Simple to code \( n^2 \) sort: \( n^2 \) comparisons, \( n \) swaps

  ```cpp
  void selectSort(tvector<string>& a) {
    int k;
    for(k=0; k < a.size(); k++) {
      int minIndex = findMin(a,k,a.size());
      swap(a[k],a[minIndex]);
    }
  }
  ```

- # comparisons: \( \sum_{k=1}^{n} k = 1 + 2 + ... + n = n(n+1)/2 = O(n^2) \)
  - Swaps?
  - Invariant: ????

Insertion Sort

- Stable sort, \( O(n^2) \), good on nearly sorted vectors
  - Stable sorts maintain order of equal keys
  - Good for sorting on two criteria: name, then age

  ```cpp
  void insertSort(tvector<string>& a) {
    int k, loc; string elt;
    for(k=1; k < a.size(); k++) {
      elt = a[k];
      loc = k; // shift until spot for elt is found
      while (0 < loc && elt < a[loc-1]) {
        a[loc] = a[loc-1]; // shift right
        loc=loc-1;
      }
      a[loc] = elt;
    }
  }
  ```

- Sorted relative to each other
  - ????
Bubble sort

- For completeness you should know about this sort
- Few (if any) redeeming features. Really slow, really, really
- Can code to recognize already sorted vector (see insertion)
  - Not worth it for bubble sort, much slower than insertion

```c
void bubbleSort(tvector<string>& a)
{ int j,k;
  for(j=a.size()-1; j >= 0; j--){
    for(k=0; k < j; k++)
      if (a[k] > a[k+1])
        swap(a[k],a[k+1]);
  }
} // "bubble" elements down the vector/array
```

Summary of simple sorts

- Selection sort has \( n \) swaps, good for “heavy” data
  - moving objects with lots of state, e.g., …
  - A string isn’t heavy, why? (pointer and pointee)
  - What happens in Java?
  - Wrap heavy items in “smart pointer proxy”
- Insertion sort is good on nearly sorted data, it’s stable, it’s fast
  - Also foundation for Shell sort, very fast non-recursive
  - More complicated to code, but relatively simple, and fast
- Bubble sort is a travesty
  - Can be parallelized, but on one machine don’t go near it

Quicksort: fast in practice

- Invented in 1962 by C.A.R. Hoare, didn’t understand recursion
  - Worst case is \( O(n^2) \), but avoidable in nearly all cases
  - In 1997 Introsort published (Musser, introspective sort)
    - Like quicksort in practice, but recognizes when it will be bad
    - and changes to heapsort

```c
void quick(tvector<string>& a, int left, int right)
{ if (left < right)
  { int pivot = partition(a,left,right);
    quick(a,left,pivot-1);
    quick(a,pivot+1, right);
  }
} // Recurrence?
```

Partition code for quicksort

- Easy to develop partition
  ```c
  int partition(tvector<string>& a, int left, int right)
  {
    string pivot = a[left];
    int k, pIndex = left;
    for(k=left+1, k <= right; k++)
      if (a[k] <= pivot)
        { pIndex++;
          swap(a[k],a[pIndex]);
        }
    swap(a[left], a[pIndex]);
  }
  ```

- loop invariant:
  - statement true each time loop test is evaluated, used to verify correctness of loop
  - Can swap into a[left] before loop
  - Nearly sorted data still ok
Analysis of Quicksort

- Average case and worst case analysis
  - Recurrence for worst case: $T(n) = \text{what about average?}$
  - Reason informally:
    - Two calls vector size $n/2$
    - Four calls vector size $n/4$
    - ... How many calls? Work done on each call?
- Partition: typically find middle of left, middle, right, swap, go
  - Avoid bad performance on nearly sorted data
- In practice: remove some (all?) recursion, avoid lots of “clones”

Merge sort: worst case $O(n \log n)$

- Divide and conquer — recursive sort
  - Divide list/vector into two halves
    - Sort each half
    - Merge sorted halves together
  - What is complexity of merging two sorted lists?
  - What is recurrence relation for merge sort as described?
  - $T(n) =$
- What is advantage of vector over linked-list for merge sort?
  - What about merging, advantage of linked list?
  - Vector requires auxiliary storage (or very fancy coding)

Tail recursion elimination

- If the last statement is a recursive call, recursion can be replaced with iteration
  - Call cannot be part of an expression
  - Some compilers do this automatically

```
void foo(int n)                 void foo2(int n) {
  if (0 < n)                      while (0 < n) {  cout << n << endl;           { cout << n << endl;
    foo(n-1);                      n = n-1;
  }
}
```

- What if `cout` << and recursive call switched?
- What about recursive factorial?

Merge sort: lists or vectors

- Mergesort for vectors
  ```
  void mergesort(vector<string>& a, int left, int right) {
    if (left < right) {
      int mid = (right+left)/2;
      mergesort(a, left, mid);
      mergesort(a, mid+1, right);
      merge(a, left, mid, right);
    }
  }
  ```
- What's different when linked lists used?
- Do differences affect complexity? Why?
- How does merge work?
Mergesort continued

- Vector code for merge isn’t pretty, but it’s not hard
  - Mergesort itself is elegant

```cpp
void merge(vector<string>& a,
           int left, int middle, int right);
// pre: left <= middle <= right,
//      a[left] <= ... <= a[middle],
//      a[middle+1] <= ... <= a[right]
// post: a[left] <= ... <= a[right]
```

- Why is this prototype potentially simpler for linked lists?
  - What will prototype be? What is complexity?

Summary of O(n log n) sorts

- Quicksort is relatively straight-forward to code, very fast
  - Worst case is very unlikely, but possible, therefore ...
  - But, if lots of elements are equal, performance will be bad
    - One million integers from range 0 to 10,000
    - How can we change partition to handle this?

- Merge sort is stable, it’s fast, good for linked lists, harder to code?
  - Worst case performance is O(n log n), compare quicksort
  - Extra storage for array/vector

- Heapsort, more complex to code, good worst case, not stable
  - Basically heap-based priority queue in a vector

Sorting in practice

- Rarely will you need to roll your own sort, but when you do ...
  - What are key issues?

- If you use a library sort, you need to understand the interface
  - In C++ we have STL and sortall.cpp in Tapestry
    - STL has sort, and stable_sort
    - Tapestry has lots of sorts, Quicksort is fast in practice
  - In C the generic sort is complex to use because arrays are ugly
    - See csort.cpp
    - In Java guarantees and worst-case are important
      - Why won’t quicksort be used?

- Function objects permit sorting criteria to change simply

In practice: templated sort functions

- Function templates permit us to write once, use several times for several different types of vector
  - Template function “stamps out” real function
  - Maintenance is saved, code still large (why?)

- What properties must hold for vector elements?
  - Comparable using < operator
  - Elements can be assigned to each other

- Template functions capture property requirements in code
  - Part of generic programming
  - Some languages support this better than others (not Java)
Function object concept

- To encapsulate comparison (like operator <) in a parameter
  - Need convention for parameter: name and behavior
  - Enforceable by templates or by inheritance (or both)
    - Sorts don’t use inheritance, `tpqueue<..>` does

- Name convention: class/object has a method named `compare`
  - Two parameters, the (vector) elements being compared
  - See comparer.h, used in sortall.h and in tpq.h
- Behavior convention: `compare` returns an int
  - zero if elements equal
  - +1 (positive) if first > second
  - -1 (negative) if first < second

Function object example

class StrLenComp // : public Comparer<string> {
  public:
    int compare(const string& a, const string& b) const {
      if (a.length() < b.length()) return -1;
      if (a.length() > b.length()) return 1;
      return 0;
    }
};

// to use this:
StrLenComp scomp;
if (scomp.compare("hello", "goodbye") < 0) …

Non-comparison-based sorts

- lower bound: \( \Omega(n \log n) \) for comparison based sorts (like searching lower bound)
- bucket sort/radix sort are not-comparison based, faster asymptotically and in practice

Shell sort

- Comparison-based, similar to insertion sort
  - Using Hibbard’s increments (see sortall.h) yields \( O(n^{3/2}) \)
  - Sequence of insertion sorts, note last value of \( h \!\!\)!”

```cpp
int k, loc, h; string elt;
h = ... // set h to 2^i-1, just less than a.size()
while (h > 0) {
  for (k = h; k < a.size(); k++) {
    elt = a[k];
    loc = h;
    while (h != 0 && elt < a[loc-h]) {
      a[loc] = a[loc-h];
      loc -= h;
    }
    a[loc] = elt;
  }
  h /= 2;
}
```