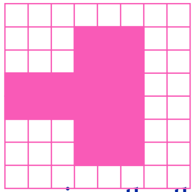
Data Compression

- Why do we care?
  - Secondary storage capacity doubles every year
  - However, disk space fills up quickly on every computer system
  - More data to compress than ever before
- What's the difference between compression for .mp3 files and compression for .zip files? Between .gif and .jpg?
- Must we exactly reconstruct the data?
  - Lossy methods
    - Generally fine for pictures, video, and audio (JPEG, MPEG, etc.)
  - Lossless methods
    - Run-length encoding

11 3 5 3 2 6 2 6 5 3 5 3 5 3 10



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

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)

```
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

	Insert O(..)	Getmin O(...)	DeleteMin O(...)
Unsorted vector			
Sorted vector			
Linked list (sorted?)			
Search tree			
Balanced tree			
Heap			

Quick look at class tpq<…>

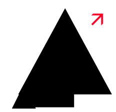
- Templated class like tstack, tqueue, tvector, tmap, ...
  - If deletemin is supported, what properties must types put into tpq have, e.g., can we insert string? double? struct?
  - Can we change what minimal means (think about anaword and sorting)?
- If we use a compare function object for comparing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

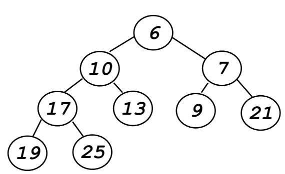
- Heap is an array-based implementation of a binary tree used for implementing priority queues, supports:
  - insert, findmin, deletemin: complexities?
- Using array minimizes storage (no explicit pointers), faster too --- children are located by index/position in array
- Heap is a binary tree with *shape* 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

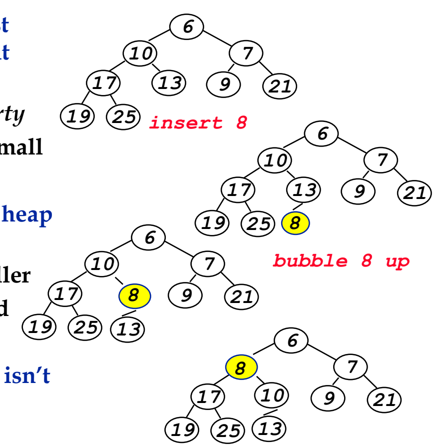
- store "node values" in array beginning at index 1
- for node with index k
  - left child: index `2*k`
  - right child: index `2*k+1`
- why is this conducive for maintaining heap shape?
- what about heap property?
- is the heap a search tree?
- where is minimal node?
- where are nodes added? deleted?

0	1	2	3	4	5	6	7	8	9	10
	6	10	7	17	13	9	21	19	25	

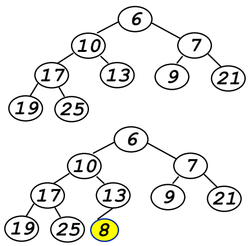


Adding values to heap

- 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



0	1	2	3	4	5	6	7	8	9	10
	6	10	7	17	13	9	21	19	25	

tvector myList

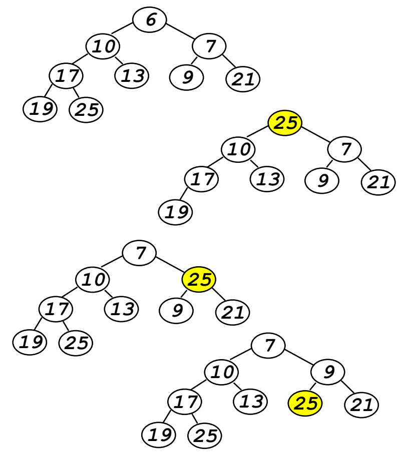
```
void pqueue::insert(int elt)
{
  // add elt to heap in myList
  myList.push_back(elt);
  int loc = myList.size();

  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"?
  -
  -

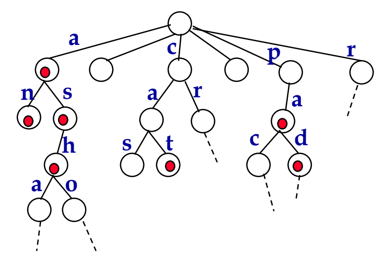


Trie: efficient search of words/suffixes

- A trie (from retrieval, but pronounced "try") supports
  - These operations are `O(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

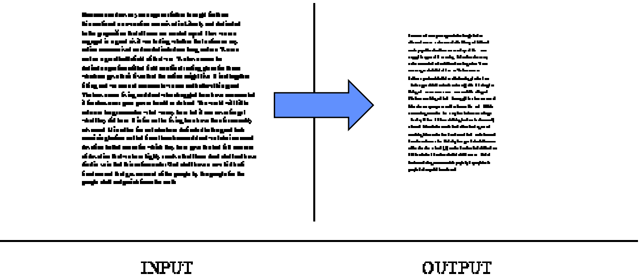
Trie picture and code (see trie.cpp)

- 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*

Text Compression



- Input: String *S*
- Output: String *S′*
  - Shorter
  - *S* can be reconstructed from *S′*

Text Compression: Examples

Encodings

ASCII: 8 bits/character

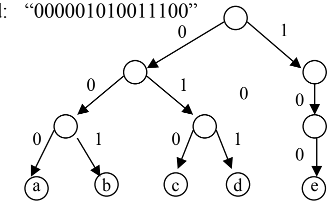
Unicode: 16 bits/character

Symbol	ASCII	Fixed length	Var. length
a	01100001	000	000
b	01100010	001	11
c	01100011	010	01
d	01100100	011	001
e	01100101	100	10

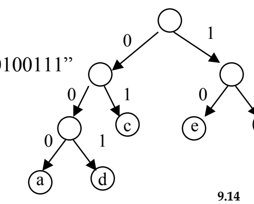
"abcd" in the different formats

ASCII: "01100001011000100110001 101100100"

Fixed: "000001010011100"



Var : "0000100111"

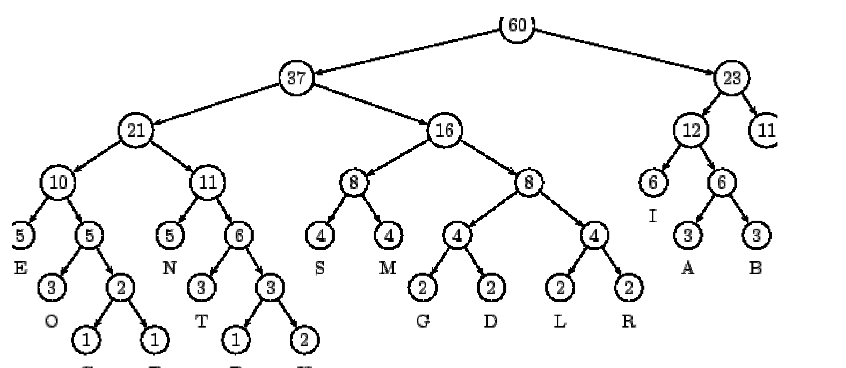


Huffman Coding

- D.A Huffman in early 1950's
- Before compressing data, analyze the input stream
- Represent data using variable length codes
- Variable length codes though *Prefix* codes
  - Each letter is assigned a codeword
  - Codeword is for a given letter is produced by traversing the Huffman tree
  - Property: *No codeword produced is the prefix of another*
  - Letters appearing frequently have short codewords, while those that appear rarely have longer ones

Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. " A SIMPLE" ⇔ "10101101001000101001110011100000"



Building a tree

- Initial case: Every character is a leaf/tree with the respective character counts → "the forest" of n trees
  n is the size of your alphabet

- Base case: there is only tree in the forest

- Reduction: Take the two trees with the smallest counts and combine them into a tree with count is equal to the sum of the two subtrees' counts
  → n-1 trees in our forest

Building a tree

"A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"

11	6	5	5	4	4	3	3	3	3	2	2	2	2	2	1	1	1
	I	E	N	S	M	A	B	O	T	G	D	L	R	U	P	F	C

Encoding

1. Count occurrence of various characters in string O( )
2. Build priority queue O( )
3. Build Huffman tree O( )
4. Write Huffman tree and coded data to file O( )

Properties of Huffman coding

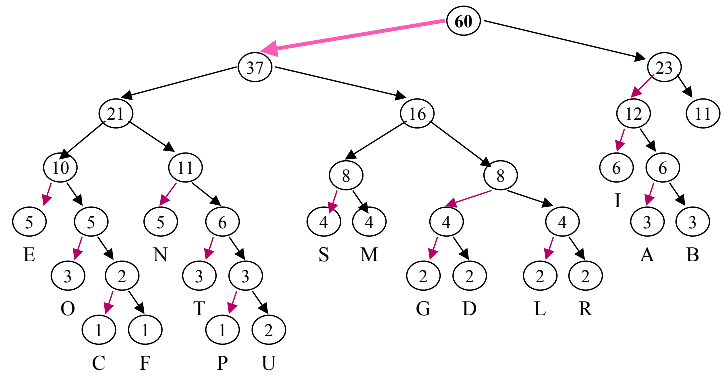
- Want to minimize weighted path length $L(T)$of tree T
- $$L(T) = \sum_{i \in Leaf(T)} d_i w_i$$
  - w_i is the weight or count of each codeword i
  - d_i is the leaf corresponding to codeword I
- Huffman coding creates pretty full bushy trees?
  - When would it produce a "bad" tree?
- How do we produce coded compressed data from input efficiently?

Writing code out to file

- How do we go from characters to codewords?
  - Build a table as we build our tree
  - Keep links to leaf nodes and trace up the tree
- Need way of writing bits out to file
  - Platform dependent?
  - UNIX `read` and `write`
- See bitops.h
  - `obstream` and `ibstream`
  - Write bits from ints
- How can differentiate between compressed files and random data from some file?
  - Store a magic number

Decoding a message

01100000100001001101



Decoding

1. Read in tree data O( )
2. Decode bit string with tree O( )

Other methods

- Adaptive Huffman coding
- Lempel-Ziv algorithms
  - Build the coding table on the fly while reading document
  - Coding table changes dynamically
  - Cool protocol between encoder and decoder so that everyone is always using the right coding scheme
  - Works darn well (`compress`, `gzip`, etc.)
- More complicated methods
  - Burrows-Wheeler (`bunzip2`)
  - PPM statistical methods

Questions

- How about ternary Huffman trees?
  - How would that affect the algorithm?
  - How about n-ary trees?
  - What would we gain?
- Are Huffman trees optimal?
  - What does that mean? (Hint: $L(T)$)
  - How can that be proven? (Hint: Induction will be your friend again)

Sorting: From Theory to Practice

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

```
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: $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = O(n^2)$
  - Swaps?
  - Invariant:

Sorted, won't move final position	?????

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

```
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

```
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]);
          }
    }
}
```

?????	*Sorted, in final position*

- "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

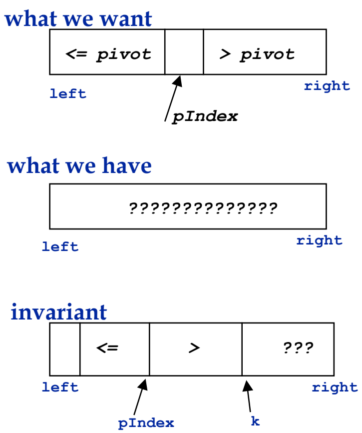
```
void quick(tvector<string>& a, int left, int right)
{
    if (left < right)
    {     int pivot = partition(a,left,right);
          quick(a,left,pivot-1);
          quick(a,pivot+1, right);
    }
}
```

- Recurrence?

<= X	X	> X

pivot index

Partition code for quicksort



- Easy to develop partition

```
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) =
  - 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"

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)
{
  if (0 < n)
  {  cout << n << endl;
     foo(n-1);
  }
}
```

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

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

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)

Merge sort: lists or vectors

- Mergesort for vectors

```
void mergesort(tvector<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

```
void merge(tvector<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
    // post: return -1/+1/0 as a.length() < b.length()
    {
        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) …
```

- We can use this to sort, see sortall.h
- Call of sort: `InsertSort(vec, vec.size(), scomp);`

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
- sort a vector of ints, all ints in the range 1..100, how?
- radix: examine each digit of numbers being sorted

23 34 56 25 44 73 42 26 10 16

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

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!!

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