COMP2160/2860: Data Structures

The University of Sydney
School of Information Technologies

Week 4:
- Red-black trees
- Sorting

2-3-4 Trees

- Rules for placing data items in the nodes of a 2-3-4 tree
  - A 2-node must contain a single data item whose search keys satisfy the relationships pictured in Figure 12-3a
  - A 3-node must contain two data items whose search keys satisfy the relationships pictured in Figure 12-3b
  - A 4-node must contain three data items whose search keys S, M, and L satisfy the relationship pictured in Figure 12-21
  - A leaf may contain either one, two, or three data items

2-3-4 Trees: Searching and Traversing a 2-3-4 Tree

- Search and traversal algorithms for a 2-3-4 tree are simple extensions of the corresponding algorithms for a 2-3 tree

2-3-4 Trees: Inserting into a 2-3-4 Tree

- The insertion algorithm for a 2-3-4 tree
  - Splits a node by moving one of its items up to its parent node
  - Splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf
  - Result: when a 4-node is split and an item is moved up to the node’s parent, the parent cannot possibly be a 4-node and can accommodate another item

2-3-4 Trees: Splitting 4-nodes During Insertion

- A 4-node is split as soon as it is encountered during a search from the root to a leaf
- The 4-node that is split will
  - Be the root, or
  - Have a 2-node parent, or
  - Have a 3-node parent

2-3-4 Trees: Splitting 4-nodes During Insertion

- Splitting a 4-node root during insertion
- Splitting a 4-node whose parent is a 2-node during insertion
2-3-4 Trees: Splitting 4-nodes During Insertion

Figure 12.30
Splitting a 4-node whose parent is a 3-node during insertion

2-3-4 Trees: Deleting from a 2-3-4 Tree

- The deletion algorithm for a 2-3-4 tree
  - Locate the node n that contains the item theItem
  - Find theItem’s inorder successor and swap it with theItem (deletion will always be at a leaf)
  - If that leaf is a 3-node or a 4-node, remove theItem
  - To ensure that theItem does not occur in a 2-node
    - Transform each 2-node encountered into a 3-node or a 4-node

2-3-4 Trees: Remarks

- Advantage of 2-3 and 2-3-4 trees
  - Easy-to-maintain balance
- Insertion and deletion algorithms for a 2-3-4 tree require fewer steps that those for a 2-3 tree
- Allowing nodes with more than four children is counterproductive

Red-Black Trees

- A 2-3-4 tree
  - Advantages
    - It is balanced
    - Its insertion and deletion operations use only one pass from root to leaf
  - Disadvantage
    - Requires more storage than a binary search tree
- A red-black tree
  - Used to represent a 2-3-4 tree
  - Has the advantages of a 2-3-4 tree, without the storage overhead

Red-Black Trees

- Basic idea
  - Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree
- Red and black children references
  - Used to distinguish between 2-nodes that appeared in the original 2-3-4 tree and 2-nodes that are generated from 3-nodes and 4-nodes
    - Black references are used for child references in the original 2-3-4 tree
    - Red references are used to link the 2-nodes that result from the split 3-nodes and 4-nodes

Red-Black Trees

- Figure 12.31
  - Red-black representation of a 4-node
- Figure 12.32
  - Red-black representation of a 3-node
Red-Black Trees: Searching and Traversing a Red-Black Tree

- A red-black tree is a binary search tree
- The algorithms for a binary search tree can be used to search and traverse a red-black tree

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Insertion algorithm
  - The 2-3-4 insertion algorithm can be adjusted to accommodate the red-black representation
  - The process of splitting 4-nodes that are encountered during a search must be reformulated in terms of the red-black representation
    - In a red-black tree, splitting the equivalent of a 4-node requires only simple color changes
    - Rotation: a reference change that results in a shorter tree
- Deletion algorithm
  - Derived from the 2-3-4 deletion algorithm

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Color changes
  - Figure 12.34
  - Splitting a red-black representation of a 4-node that is the root

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Color changes
  - Figure 12.35
  - Splitting a red-black representation of a 4-node whose parent is a 2-node

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Color changes
  - Figure 12.36a
  - Splitting a red-black representation of a 4-node whose parent is a 3-node

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Color changes
  - Figure 12.36b
  - Splitting a red-black representation of a 4-node whose parent is a 3-node
Red-Black Trees: Inserting and Deleting From a Red-Black Tree

Figure 12.36c
Splitting a red-black representation of a 4-node whose parent is a 3-node

Sorting algorithms
- Bubblesort
- Insertionsort
- Selectionsort
- Mergesort
- Quicksort
- Shellsort
- Heapsort
- Bucket sort

Bubble Sort

- Bubble sort
  - Strategy
    - Compare adjacent elements and exchange them if they are out of order
    - Comparing the first two elements, the second and third elements, and so on, will move the largest (or smallest) elements to the end of the array
    - Repeating this process will eventually sort the array into ascending (or descending) order

Analyzing bubble sort

```plaintext
array elements[1..N]
for j := 1 to N do
  for k := j+1 to N do
    if elements[k] > elements[k+1]
      then swap(k, k+1, elements)
    end // if
  end // for k loop
end // for j loop
```
Different bubble-sort

array elements[1..N]
swapDone = true
while swapDone do
    swapDone = false
    for k := 1 to N-1 do
        if elements[k] > elements[k+1] then
            swap(k, k+1, elements)
            swapDone = true
        end // if
    end // for k loop
end // while loop

Bubble-sort

- Need to look at the algorithm
- Find properties for finishing the loop
  - no swaps done
- Why is the worst case complexity still $O(n^2)$?
- Is this a better solution than nested for loops?

Insertion Sort

- Insertion sort
  - Strategy
    - Partition the array into two regions: sorted and unsorted
    - Take each item from the unsorted region and insert it into its correct order in the sorted region

  Figure 9.6
  An insertion sort partitions the array into two regions

  ![Insertion Sort](image)

Insertion Sort Analysis

- Worst case: $O(n^2)$
  - For small arrays
    - Insertion sort is appropriate due to its simplicity
  - For large arrays
    - Insertion sort is prohibitively inefficient

Selection Sort

- Find next largest element in the sequence and place it in the correct position
  - Worst case: $O(n^2)$
**Elementary sorts**

- Insertion, Selection, bubblesort
- All require
  - quadratic time
  - worst case and on the average
  - no extra space

**Shellsort**

- Extension to insertion sort
- Exchange items far apart
- $h$-sorted sequence
  - taking every $h$-th element yields a sorted subsequence
  - every $h$-th element, starting anywhere
- $h$-sorted sequence is $h$ independent sorted sequences put together

**Shellsort**

- Main idea:
  - Sort for large values of $h$
    - this allows long-distance swaps
    - proceed with smaller $h$ values
    - until $h=1$
- For every $h$-pass, use insertion sort
  - on the $h$ subsequence

**Shellsort**

- Choosing an increment sequence for $h$
  - hard problem
- In practice we choose a sequence that decreases geometrically
- A good $h$-sequence can give up to 25% speedup
- Amazing theoretical results concerning shellsort sequences

**Shellsort**

- Increments 1 4 13 40 121 364 1093 
  3280 9841 ... lead to $O(n^{3/2})$ shellsort
- 1 8 23 77 281 1073 4193 16577 ... lead to $O(n^{4/3})$ shellsort
  - Increments are relatively prime
- 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81...
  - lead to a $O(N(\log N)^2)$ shellsort
  - triangle values

**Mergesort**

- Important divide-and-conquer sorting algorithms
  - Mergesort
  - Quicksort
- Mergesort
  - A recursive sorting algorithm
  - Gives the same performance, regardless of the initial order of the array items
  - Strategy
    - Divide an array into halves
    - Sort each half
    - Merge the sorted halves into one sorted array
Mergesort

Mergesort analysis

- Mergesort running time

\[
T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
T(n/2) + T(n/2) + n & \text{otherwise} 
\end{cases}
\]

Mergesort

Quicksort

- Analysis
  - Worst case: \(O(n \times \log_2 n)\)
  - Average case: \(O(n \times \log_2 n)\)
  - Advantage
    - It is an extremely efficient algorithm with respect to time
  - Drawback
    - It requires a second array as large as the original array

Quickstart

- Quickstart
  - A divide-and-conquer algorithm
  - Strategy
    - Partition an array into items that are less than the pivot and those that are greater than or equal to the pivot
    - Sort the left section
    - Sort the right section

Figure 9.12

A partition about a pivot
Quicksort

- Using an invariant to develop a partition algorithm
  - Invariant for the partition algorithm
    The items in region $S_1$ are all less than the pivot, and those in $S_2$ are all greater than or equal to the pivot.

![Invariant for the partition algorithm](image)

Analysis

- Worst case
  quicksort is $O(n^2)$ when the array is already sorted and the smallest item is chosen as the pivot.

![A worst-case partitioning with quicksort](image)

- Average case
  quicksort is $O(n \times \log_2 n)$ when $S_1$ and $S_2$ contain the same or nearly the same number of items arranged at random.

![A average-case partitioning with quicksort](image)

- Analysis
  quicksort is usually extremely fast in practice.
  - Even if the worst case occurs, quicksort’s performance is acceptable for moderately large arrays.

Radix Sort

- Radix sort
  - Treats each data element as a character string.
  - Strategy
    - Repeatedly organize the data into groups according to the $i$th character in each element.

- Analysis
  - Radix sort is $O(n)$.
A Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Sort Algorithm</th>
<th>Worst case</th>
<th>Average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Quick sort</td>
<td>$n^2$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$n^2$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
</tbody>
</table>

Approximate growth rates of time required for eight sorting algorithms.

Figure 9.22

A lower bound for sorting

- Comparison based sorting
  - $\Omega(n \log n)$ lower bound
  - any comparison based sort algorithm will require at least $n \log n$ comparisons

Lower Bound (cont’d)

- Assume we have only the operations
  - Compare two elements
  - Swap elements
- Can we do better than $O(n \log n)$, e.g. heapsort and mergesort?
- How to prove that there exists no better algorithm than $O(n \log n)$?

Lower Bound (cont’d)

- Proof: we have to prove that no algorithms can perform better
  - Construct a model for all comparison based sorting algorithms
  - Model is a decision tree
    - Root node: input sequence
    - Internal node: partially sorted sequence
    - Leaf node: sorted sequence
    - Binary: decision based on comparison
  - Worst-Case: height of the tree

Lower Bound (cont’d)

- Assume we have a sequence a,b,c and all elements have different values
- Decision tree of the sequence:

```
          a < b < c
        /    |    \
      V     V     V
    a < b < c
   /   |   \
  a < b
 /   |   \  
 a < b < c < a < b < c < a < b < c
```

Lower Bound (cont’d)

- Sorting: generate a permutation for a sequence
- Any sequence as input: therefore any permutation as output possible
- Correctness: generate all possible outputs.
- Therefore, in decision tree there must be one leaf for every permutation.
- We have $n!$ permutations, therefore $n!$ leaves
- A binary tree with $n!$ leaves must have at least $\log(n!)$ depth
Lower Bound (cont’d)

• Use Sterling’s formula

\[ n! = \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left( 1 + O\left(\frac{1}{n}\right) \right) = \Omega\left(n^n\right) \]

\[ \log(n!) = \Omega\left(\log n^n\right) = \Omega\left(n \log n\right) \]