Data Structures and Algorithms Midterm

2017-02-21

- **Stack** is also called a **Last-In-First-Out (LIFO)** data structure, whereas **queue** is called a **First-In-First-Out (FIFO)** data structure.
- Operations on stack: initialize, pop, push, full, empty, peek.
- Applications of stack: recursive function calls, system stack.
- A system stack consists of a pointer to the previous frame, return address, and local variables.
- Example: implementation of stack ArrayStack.c
- Example: implementation of queue ArrayQueue.c
- Practice: implement a calculator and a maze solver using stack.
- [x] Homework: http://www.csie.ntu.edu.tw/~hsinmu/courses/_media/dsa_17spring/r1.pdf

2017-03-07

- **Time complexity T(n)**: the time required to complete the execution of the entire algorithm/program.
- **Space complexity S(n)**: the space (memory) required to complete the execution of the entire algorithm/program.
- **Input size** can be the size of the array, the dimensions of a matrix, the exponent of the highest order, or the number of bits in a binary number.
- Running time can be considered in the *worst case*, *average case*, and the *best case*.
- Average case is often *as bad as* the worst case.
- Running time comparison: *constant* < *logarithm* < *linear* < *log-linear* < *quadratic* < *cubic* < *exponential*.
- [x] Homework: dsa_2017_hw1_3.pdf
- Solutions: Solutions1.pdf, calculator.c, good_string.c

2017-03-14

- Running time can be denoted as:
  - O: upper bound. $\exists k > 0 \; \exists n_0$ such that $f(n) \leq k \cdot g(n) \; \forall n > n_0$
  - Θ: tight bound. $\exists k_1 > 0 \; \exists k_2 > 0 \; \exists n_0$ such that $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n) \; \forall n > n_0$
  - Ω: lower bound. $\exists k > 0 \; \exists n_0$ such that $k \cdot g(n) \leq f(n) \; \forall n > n_0$

- o: strict upper bound. $\forall k > 0 \; \exists n_0$ such that $f(n) < k \cdot g(n) \; \forall n > n_0$
- ω: strict lower bound. $\forall k > 0 \; \exists n_0$ such that $k \cdot g(n) < f(n) \; \forall n > n_0$

- The property of $f(n) = \mathrm{O}(g(n))$ holds after some operations: *summation*, *multiplication*, and *power*, but does not hold after the following operations: *logarithm*, and *exponential*.
- **Linked list** can be implemented with array or by creating a real pointer pointing to the next *node*.
  - **Singly** v.s. **doubly** linked list
  - **Circular** v.s. **non-circular** linked list
- A brief list of abstract data type (ADT): *bag (container), graph, list, map (associative array, dictionary, hash table), queue, set, stack, tree*.
- Every ADT can be implemented with either arrays or linked structures.
- Creating a virtual head pointing to the real head of the data can reduce the complexity of the code of linked list.
- Linked list wastes $O(n)$ to store the address of the next node.
- **Doubly linked list** makes accessing the tail node much more easily (if we are not going to use a tail pointer or a circular linked list).
- How to make a memory-efficient doubly linked list? http://goo.gl/qifrq2
- Practice:
  - Given a (singly) linked list of unknown length, design an algorithm to find the n-th node from the tail of the linked list. Your algorithm is allowed to traverse the linked list only once.
  - Reverse a given singly linked list using the original link nodes.
- [x] Homework: http://goo.gl/qifrq2

2017-03-21

- Recycling costs $O(n)$ time complexity to free all nodes.
- An alternative to recycling is to collect all "deleted nodes", and use them when necessary.
- **Circular linked list** makes inserting a cluster of nodes into another linked list more efficient (if we are not going to use a tail pointer or a doubly linked list).
- Example: implementation of circular doubly linked list with efficient memory LinkedList.c
- **String-matching** problem: given a *pattern P* and *text T*, find pattern P occurs with **shift** s in text T.
- Definitions:
  - $|x|$: the length of the string x
  - xy: concatenation of two strings x and y
  - $P \sqsubset T$: P is the **prefix** of T
  - $P \sqsupset T$: P is the **suffix** of T

- **Overlapping-suffix lemma**: Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$.
  - If $|x| \leq |y|$, then $x \sqsupset y$.
  - If $|x| \geq |y|$, then $y \sqsupset x$.
  - If $|x| = |y|$, then $x = y$.
- **Native string matcher**:
  - Preprocessing time: 0
  - Matching time: $O((|T| - |P| + 1)|P|) = O(|T||P|)$
- **Knuth-Morris-Pratt (KMP) algorithm**:
  - Preprocess the pattern with **prefix function**, or called **failure function**, which calculates the length of the longest prefix which is also a postfix at each position, and returns **prefix table**, or **fail table**, or **partial match talbe**.
  - Take advantage of the preprocessed table and scan through the text to look for any matches.
  - Preprocessing time: $\Theta(|P|)$
  - Matching time: $\Theta(|T|)$

2017-03-28

- **Rabin-Karp algorithm**:
  - Convert the pattern and every substring of the text to a hash value and then match the hash value.
  - It is possible that the hash value is too large to fit a `int` or even `uint64_t` data type, so we may have to make computed hash value modulo a suitable *modulus* q.
  - If a mod q equals b mod q, it is notated as $a \equiv b(\text{mod } q)$.
  - However, it is possible that two different strings have the same remainder. Therefore, Any shift s for which $a \equiv b(\text{mod } q)$ must be tested further to see whether s is really valid or merely a **spurious hit**. If this happen, it takes addition $O(|P|)$ to perform the extra checking.
  - Thankfully, *spurious hits* occur infrequently enough that the cost of the extra checking is low.
  - Preprocessing time: $\Theta(|P|)$
  - Matching time: $O((|T| - |P| + 1)|P|)$ = $O(|T||P|)$
- Some terms related to sorting:
  - **Internal** v.s . **external**: *internal* sorting places all data in the memory, while *external* sorting occurs when the data is too large to fit entirely in the memory, which necessitates the use of other (slower) storage, e.g., hard drive, flash disk, network storage, etc.
  - **In-place**: directly sorts the keys at their current memory locations.
  - **Stable**: if a_i and a_j have equal key value, they maintain the same order before and after sorting.

- **Adaptive**: If *part of the sequence is sorted*, then the time complexity of the sorting algorithm reduces.
- **Online**: the ability to sort and take input at the same time.
- How much time do we need in the *best case* and *worst case* for sorting?
  - The *best-case* time complexity is (at least) $\Omega(n)$
  - Any *comparison-based* sorting algorithms have *worst-case* time complexity of (at least) $\Omega(n \log n)$.
- Decision tree for sorting n items:
  - A *binary tree* with each node representing a comparison & swap.
  - There are $n!$ leaves since there are that many possible permutations.
  - In the *worst case*, the time it takes to sort is the height of the binary tree.
- **Selection sort**: best-case, average-case and worst-case are all $O(n^2)$, space complexity is $O(1)$, (+) in-place, (-) stable, (-) adaptive.
- **Insertion sort**: best-case is $O(n)$, average-case and worst-case are both $O(n \log n)$, space complexity is $O(1)$, (+) in-place, (+) stable, (+) adaptive.
  - Use binary search can reduce time to look for the location to insert.
  - Use linked list to store the items can reduce time to insert an item.
  - The best case is when the array is already sorted.
  - The worst case is when the array is reversely sorted.
- **Merge sort**: best-case, average-case and worst-case are all $O(n \log n)$, space complexity is $O(n)$, (-) in-place, (+) stable, (-) adaptive.
  - Use *divide-and-conquer* strategy
- **Quick sort**: best-case and average-case both $O(n \log n)$, worst-case is $O(n^2)$, space complexity is $O(\log n)$, (+) in-place, (-) stable, (-) adaptive.
  - Use *divide-and-conquer* strategy
  - The best case is when the array is partitioned to equal half each round.
  - The worst case is when the array is sorted or reversely sorted.
- [x] Homework: dsa_2017_hw2_1.pdf
- Solutions: b00401062_hw2.pdf, string_pair.c, secret_code.c

2017-04-11

- **Tree**: a tree is a finite set of one or more nodes such that
  - There is a specially designated node called the **root**.
  - The remaining nodes are partitioned into $n \geq 0$ disjoint sets, which are called the **subtrees** of

the root.

- Terminology for tree:
  - **Degree (of a node)**: the number of subtrees of a node
  - **Level**: the number of branches to reach that node from the root node
  - **Height/Depth**: the number of levels in a tree
  - **Size**: the number of nodes in a tree
  - **Weight**: the number of leaves in a tree
  - **Degree (of a tree)**: the maximum degree of any node in a tree
- Tree can be represented with *array*. Assume the degree (of the tree) is d
  - For node with index i, its parent's index is $(i-1)/d$.
  - For node with index i, its children's indices range from $i \cdot d + 1$ to $i \cdot d + 3$
- Representing a tree with linked structure:
  - Assume the degree of the tree = d, the size of the tree = n, then the number of null pointers in the tree is total number of pointers - the number of branches = $nd - (n-1)$
  - **Left child-right sibling (LCRS) representation**: similar to binary tree such that one pointer points to a *leftmost child*, and the other pointer points to a *immediately-right sibling*.
  - Root does not have a right child in LCRS because root does not have a sibling in the original tree.
- **Binary tree**: a binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the **left subtree** and the **right subtree**.
- Some properties about binary tree:
  - The number of nodes at level i is at most 2^i.
  - A tree of height h has at most $2^h - 1$ nodes.
  - Given a tree with n_0 leaf nodes and n_2 degree-2 nodes, then $n_0 = n_2 + 1$. Proof: $n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$
- **Full binary tree**: a binary tree of height h having $2^h - 1$ nodes, i.e. all nodes except leaves have two children.
- **Complete binary tree**: a binary tree with n nodes and height h is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of height h.
- Given a complete binary tree with n nodes, the height of the tree is the ceiling of $\log_2(n+1)$
- Binary tree traversal has 3 variations: **VLR (preorder), LVR (inorder), LRV (postorder)**. Traversal can be achieved through recursive method or non-recursive methods together with *stack*.
- Arithmetic expression represented with binary tree can be **prefix, infix, postfix**.
- **Binary search tree (BST)**: a binary tree whose left subtrees have smaller keys and right subtrees have larger keys.-

- How to insert a node? insert directly!
- How to delete a node?
    - Leave: delete directly!
    - Degree-1 node: attach its only child to its parent.
    - Degree-2 node: find *the largest node of the left subtree* or *the smallest of the right subtree* and move it to the node to be deleted.