Linked List

- Easy insertion at head and tail
- Also easy insertion at middle if already know where the node is
- Sequential access (in contrast to array's random access)
- Application: sparse array

Complexity

- $O(g(n)), \Omega(g(n)), \Theta(g(n))$
- Common $g(n)$:
    - 1 (constant)
    - $\lg(n)$ (logarithmic)
    - n (linear)
    - $n\lg(n)$
    - n^2
    - n^k (polynomial)
    - 2^n (exponential)
- Properties
    - Cares about large n
    - 封閉律: if $f_1(n) = O(g(n)),\ f_2(n) = O(g(n))$ then $f_1(n) + f_2(n) = O(g(n))$
    - 併吞律: if $f_1(n) = O(g_1(n)),\ f_2(n) = O(g_2(n)),\ g_1(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(g_2(n))$
    - if $f(n)$ is a polynomial with highest degree k then $f(n) = O(n^k)$

Stack

- LIFO
- Operations
    - `push()`
    - `pop()`
    - `peek()`
- DFS
- Application: expression evaluation
    - Infix notation $\Rightarrow$ Postfix notation
- Application: expression parsing

Queue

- FIFO

- Operations
  - `enqueue()`
  - `dequeue()`
- BFS
- Variation: Deque
  - two-way queue/stack

Tree

- Keywords
  - root, internal node, leaf
  - subtree
  - parent, child
- Translate general tree to binary tree
  - left-child, right-sibling
- Full binary tree
  - easily packed into array

Heap

- Max-heap
  - tree-like structure
  - root has the largest value
  - subtrees also satisfy max-heap properties
- Heap sort
  - build heap $\rightarrow$ use heap to select largest
  - time complexity:$O(n \lg(n)) + O(n \lg(n))$
  - build heap can be in linear time if optimizly implemented

---

Sorting

- Selection sort
  - select minimum value from "unsorted" part and put it in "sorted" part
  - extra space: $O(1)$
  - time: $\Theta(n^2)$
  - unstable with common implementation, but can be stable with rotating
- Heap sort
  - selection sort + max-heap
  - extra space: $O(1)$
  - time: $O(n \lg(n))$

    - unstable
- Insertion sort
    - insert element into "sorted" part
    - extra space: $O(1)$
    - time: $O(n^2)$
    - stable
    - adaptive
- Shell sort
    - insertion sort for every k_1 step, then k_2, then $\cdots$
    - extra space: $O(1)$
    - time: usually better than $O(n^2)$, depends on choose of k
    - unstable
    - adaptive
    - easy to implement and decent performance
- Merge sort
    - time: $O(n \lg(n))$
    - can be stable if implemented carefully
    - can be parallelize
    - popular in external sort
- Quick sort
    - extra space: average $O(\lg(n))$, worst $O(n)$
    - time: average $O(n \lg(n))$, worst $O(n^2)$
    - unstable
    - usually the best choice for large data (if stability isn't concerned)

Some Code

Binary Search

```
function Binary_Search(A, key)
    left = 1, right = A.len
    while left <= right
        mid = (left+right)//2
        if A[mid] == key
            return mid
        else if A[mid] < key
            left = mid+1
        else if A[mid] > key
            right = mid-1
    return None
```

Parentheses Matching

```
for c in input
    if c is left
        push(c)
    else if c is right
        d = pop()
        check if c and d match
```

Eval

```
for token in input
    if token is number
        push(token)
    else if token is operator
        n1 = pop(), n2 = pop()
        push(operate(n1, n2, token))
return pop()
```

Fix-translation

```
for token in input
    if token is number
        print(token)
    else if token is operator
        while peek() >= token
            print(pop())
        push(token)
```

BST-search

```
function BST_Search(T, key)
    node = T.root
    while node != NULL
        if node.key == key
            return node
        else if node.key < key
            node = node.right
        else if node.key > key
            node = node.left
    return None
```

Quick sort

```
function QuickSort(a)
    pivot = a[0]
    for i from 1 to n-1
        if a[i] < pivot
            left.append(a[i])
        else
            right.append(a[i])
    QuickSort(left)
    QuickSort(right)
    a = left + pivot + right
```