Array and Linked List
Array

- Array of size $n$: stored in $n$ contiguous memory space
- Access any element in the array: $O(1)$ time
Array of size $n$: stored in $n$ contiguous memory space
Access any element in the array: $O(1)$ time
Drawbacks:
- Need to reallocate memory when the size is changed
- Insert element(s) to the array needs $O(n)$ time
Each element stores a data and a pointer to the next element.
Linked List

- Each element stores a data and a pointer to the next element

![Diagram of a linked list](image)

- No need to reallocate memory when the size is changed
- Insertion: no need to copy $O(n)$ elements
- However, accessing the $i$-th element requires $O(i)$ time (worst case $O(n)$)
Dynamic Array

- Goal: design an array with changeable size
Dynamic Array

- Goal: design an array with changeable size
- A Dynamic Array:
  - An array of data \( a \)
  - \( a.size \): the current size of array
  - \( a.capacity \): the memory allocated for the array

When inserting an element to the end:
- If \( a.capacity > a.size + 1 \): directly put the element in the end of array
- If \( a.capacity == a.size + 1 \):
  1. double the capacity
  2. reallocate \( a \) space with the new capacity
  3. copy the contents to the new memory location
Dynamic Array

- Goal: design an array with changeable size
- A Dynamic Array:
  - An array of data $a$
  - $a.size$: the current size of array
  - $a.capacity$: the memory allocated for the array
- When inserting an element to the end:
  - If $a.capacity > a.size + 1$: directly put the element in the end of array
  - If $a.capacity == a.size + 1$: (1) double the capacity (2) reallocate a space with the new capacity (3) copy the contents to the new memory location
Dynamic Array

Capacity=4, size=3

| 0 | 9 | 26 |

Insert 31 to the end

Capacity=4, size=4

| 0 | 9 | 26 | 31 |

Insert 41 to the end

Capacity=8, size=5

| 0 | 9 | 26 | 31 | 41 |

Insert 42 to the end

Capacity=8, size=6

| 0 | 9 | 26 | 31 | 41 | 42 |
Python List

- Python List: a dynamic array of pointers (to support different objects in the list)
- Check [http://www.laurentluce.com/posts/python-list-implementation/](http://www.laurentluce.com/posts/python-list-implementation/)
Data Structure: Priority Queue
Priority Queue

Priority queue is a data structure that stores a set of objects, and supports two basic operations:

- **insert** (enqueue): insert a new object to the priority queue
- **delete-min** (dequeue): finds the current **minimum** element, delete it from queue, and return it.

Usually a priority queue also supports: find-min, clean, change one element

![Priority Queue Diagram]

Dequeues the next element with the highest priority
A naive implementation of priority queue using sorted list:
- $O(n)$ insert
- $O(1)$ delete-min
A naive implementation of priority queue using sorted list:
- $O(n)$ insert
- $O(1)$ delete-min

**Heap** is one of the most useful (and simple) priority queue

It support
- $O(\log n)$ time per insert
- $O(\log n)$ time per delete-min
- $O(1)$ time for find-min
- $O(\log n)$ time for change one element
A heap is a complete binary tree with $n$ nodes:
- Only the bottom most level may be partially filled (from left to right)
- Therefore, height is $O(\log n)$

In a heap, each node is larger than its parent (except root, which has no parent)

No other ordering rules $\Rightarrow$ there can be multiple heaps for the same data
Heap: array representation

- Heap can be easily stored in an array (because it is a full binary tree!)
- Traverse the heap from node $i$
  - Left child: node $2i + 1$
  - Right child: node $2i + 2$
  - Parent: node $i/2$
Heap: find min

- How to get the minimum of the heap?
  The minimum is always at the root!
- Only takes $O(1)$ (constant) time.
Heap: insertion

- Insert a new element to the heap.
- Need to keep the constraints that each node is greater or equal to its parent.
- Use the operation called “heapify”
Heap: insertion

- Compare the current element and its parent; swap if they violate the ordering
- Worst case: go from leaf to root ⇒ $O(\log n)$ time

```python
### add value v to heap A
A.append(v)
i = len(A)-1
while (i>0 & & A[i/2]> A[i]):
    (A[i/2],A[i]) = (A[i],A[i/2])
i = i/2
```
Heap: insertion
Return and remove the root element.

Need to maintain the heap structure:
- Move the last element to the root
- Heapify (adjust ordering from root to a leaf)
- \(O(\log n)\) time

```python
### remove the root element from A
A[0] = A[-1]
del A[-1]
i = 0
while (i<len(A)):
    ## Need to consider boundary cases in practice
        break
    if (A[i*2+1] < A[i*2+2]):
        (A[i], A[i*2+1]) = (A[i*2+1], A[i])
        i = i*2+1
    else:
        (A[i], A[i*2+2]) = (A[i*2+2], A[i])
        i = i*2+2
```
Using heap for sorting

- Insert all the elements to the heap
  \[ O(n \log n) \]
- Extract and remove the minimum at a time (total \( n \) times)
  \[ O(n \log n) \]
- Heap-sort: \( O(n \log n) \) time complexity
Data Structure: Binary Search Tree
Data Structure that supports “search”

- A table of records in which a key is used for retrieval.
  
  key1:value1  key2:value2  ...  keyn:valuen

- Store in an array:
  - $O(n)$ search time (in the worst case, go through the whole array)

- Can we have a better structure to improve the search time?
Binary Search Tree

- Binary search tree property:
  - The key in each node $\geq$ any key stored in the left sub-tree
  - The key in each node $\leq$ any key stored in the right sub-tree
Searching for a key

Given a key $k$, search for the node with this key

- For node $i$, if $key[i] > k$
  
  Only need to search for the left subtree

- For node $i$, if $key[i] < k$

  Only need to search for the right subtree
Searching for a key

- Go from root to leaf.
- Time: proportional to the height of the tree
  $O(\log n)$ if the tree is balanced.

![Tree Diagram]

Search for 4
Insert a key

- Step 1. Find the insert location
  \( O(\log n) \) time using BST search algorithm
  (assume height = \( O(\log n) \))
- Step 2. Insert the node: constant time
Insert a key

- **Step 1.** Find the insert location
  
  \( O(\log n) \) time using BST search algorithm
  
  (assume height = \( O(\log n) \))

- **Step 2.** Insert the node: constant time

- **Step 3.** Balance the tree

  Various ways (AVL tree, Red black tree, etc)
  
  Sublinear time (often \( O(\log n) \))
### Overall Time Complexity

**AVL Tree (Balanced Binary Tree):** guaranteed $O(\log n)$ tree height
- Space: $O(n)$
- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$

**Red Black Tree:** guaranteed $O(\log n)$ tree height
- Space: $O(n)$
- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$
Data Structure: Hash Tables
Hash Tables

- A data structure that supports **insert**, **search**, **delete** in $O(1)$ time (in expectation)
- Store the key-value pairs:
  - key1:value1   key2:value2   ...   keyn:valuen
Hash Tables

- A data structure that supports **insert**, **search**, **delete** in $O(1)$ time (in expectation)
- Store the key-value pairs:
  
  \[
  \text{key1:value1 \hspace{1cm} key2:value2 \hspace{1cm} \ldots \hspace{1cm} keyn:valuen}
  \]

- Basic idea: save items in a key-indexed table (index is a function of the key)
- **Hash function**: method for computing array index from key

\[
\begin{align*}
\text{hash(103)} &= 3 \\
\text{hash(4571)} &= 1
\end{align*}
\]
Hash Tables

Issues:

- Designing the hash function $f$ to map keys (input domain) to index (output domain)
- Collision Resolution: how to handle two keys that hash to the same index?
Hash Tables

Issues:

- Designing the hash function $f$ to map keys (input domain) to index (output domain)
- Collision Resolution: how to handle two keys that hash to the same index?

Space-time tradeoff: Designing the function and size of hash table:

- Large hash table: more memory but less time (less confliction)
- Smaller hash table: less memory but more time (more confliction)
Designing the Hash Function

- Ideal goal: Scramble the keys uniformly to produce a table index
  - Efficiently computable
  - Each table index equally likely for each key

Commonly used hash functions:
- Modulo by integer (usually prime). E.g., key%27 for 27 buckets
- MD5
- String: transform to integer first:

\[ h = s[0] \cdot 26^{L-1} + \ldots + s[L-2] \cdot 26^1 + s[L-1] \cdot 26^0 \]
Collisions

- Collision: Two distinct keys hashing to the same index
- Can be handled by a hash table of lists (each index stores a linked list)
  - Hash: map key to integer $i$ between 0 and $M - 1$
  - Insert: insert to the $i$-th chain
  - Search: need to search only $i$-th chain
Time and Space Complexity

- **Worst case:**
  - Search: $O(n)$
  - Insert: $O(n)$
  - Delete: $O(n)$

- **Under uniform hashing assumption:**
  - Search: $O(1)$ in average
  - Insert: $O(1)$ in average
  - Delete: $O(1)$ in average

- In python, dictionary is implemented by hash table
Two-Sum: A Linear Time Algorithm

- Insert everything in the hash table
- For each $a[i]$, check whether $(Target - a[i])$ is in the hash table
- $O(n)$ time in average

```python
def two_sum(a, T):
    n = len(a)
    mytable = {}
    for i in range(n):
        if (T-a[i]) in mytable:
            return (mytable[T-a[i]], i)
        mytable[a[i]] = i
    return -1
```
Python Data Structure: Sparse Matrix
Dense matrix Storage

- When storing the elements of a 2-D array in memory, these are allocated contiguous memory locations
  $\Rightarrow$ A 2-D array must be linearized to 1-D in storage
- Dense matrix can be vectorized by column major or row major
- (numpy array is in row major)
Dense Matrix Operations

- Let $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m \times n}$, $s \in \mathbb{R}$
  
  $A + B$, $sA$, $A^T$: $mn$ operations

- Let $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^{n \times 1}$
  
  $Ab$: $mn$ operations
Matrix-matrix multiplication:
what is the time complexity of computing $AB$?
Dense Matrix Operations

- Assume $A, B \in \mathbb{R}^{n \times n}$, what is the time complexity of computing $AB$?
- Naive implementation: $O(n^3)$
- Theoretical best: $O(n^{2.xxx})$ (but slower than naive implementation in practice)
- Best way in practice: using BLAS (Basic Linear Algebra Subprograms)
Dense Matrix Operations

- BLAS matrix product: $O(mnk)$ for computing $AB$ where $A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}$
- Compute matrix product block by block to minimize cache miss rate
- Can be called from C, Fortran; can be used in MATLAB, R, Python, ... 
- Three levels of BLAS operations:
  1. Level 1: Vector operations, e.g., $y = \alpha x + y$
  2. Level 2: Matrix-Vector operations, e.g., $y = \alpha A x + \beta y$
  3. Level 3: Matrix-Matrix operations, e.g., $y = \alpha A B + \beta C$
BLAS
Dense Matrix vs Sparse Matrix

- Any matrix $X \in \mathbb{R}^{m \times n}$ can be stored as dense or sparse
- Dense Matrix: most entries in $X$ are nonzero ($mn$ space)
- Sparse Matrix: only few entries in $X$ are nonzero ($O(nnz)$ space)
Sparse Matrix

- Widely-used format: Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), ...
- CSC: three arrays for storing an $m \times n$ matrix with $nnz$ nonzeros
  1. val ($nnz$ real numbers): the values of each nonzero elements
  2. row_ind ($nnz$ integers): the row indices corresponding to the values
  3. col_ptr ($n + 1$ integers): the list of value indexes where each column starts

![Matrix Visualization]
Sparse Matrix

- Widely-used format: Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), ...
- CSR: three arrays for storing an $m \times n$ matrix with $nnz$ non-zeroes
  - val ($nnz$ real numbers): the values of each non-zero element
  - col_ind ($nnz$ integers): the column indices corresponding to the values
  - row_ptr ($m + 1$ integers): the list of value indexes where each row starts

![Diagram of CSR arrays]

<table>
<thead>
<tr>
<th>row_ptr</th>
<th>col_idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>60</td>
</tr>
</tbody>
</table>
The CSR format in scipy.

Can be constructed from dense matrix, sparse matrix, shape tuple (empty matrix), \((i, j, v)\) pairs, or the three arrays for CSR.

```python
>>> import scipy.sparse as sparse
>>> mtx = sparse.csr_matrix((3, 4))
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
>>> mtx = sparse.csr_matrix(([10,10], ([1, 2], [2, 3])))
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 10, 0],
        [0, 0, 0, 10]])
```
Sparse Matrix in Python

```python
>>> mtx.data
array([10, 10])
>>> mtx.indices
array([2, 3], dtype=int32)
>>> mtx.indptr
array([0, 0, 1, 2], dtype=int32)
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> indptr = np.array([0, 2, 3, 6])
>>> mtx = sparse.csr_matrix((data, indices, indptr))
>>> mtx.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```
The CSC format in scipy.
Can be constructed from dense matrix, sparse matrix, shape tuple
(empty matrix), \((i, j, v)\) pairs, or the three arrays for CSC.

```python
>>> import scipy.sparse as sparse
>>> mtx = sparse.csc_matrix((3, 4))
>>> mtx.todense()
matrix([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=int8)
>>> mtx = sparse.csc_matrix(((10, 10), ([1, 2], [2, 3])))
matrix([[0, 0, 0, 0],
        [0, 0, 10, 0],
        [0, 0, 0, 10]])
```
Questions?