Lecture 7: Basics of Data Structures

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Bhupat and Jyoti Mehta School of Biosciences
Indian Institute of Technology Madras

August 28, 2018

Introduction

Contiguous vs. Linked Data Structures

Linked

- Extra storage required
- Better use of fragmented memory
- Insertion/deletion at middle is easier
- Joining lists easier
- 'Next' operation requires pointer dereference

Contiguous

- Next and Previous are implicit (less storage)
- Can take advantage of locality
- Random access
- 'Next' operation probably faster

Contiguous vs. Linked Data Structures

Linked

- Extra storage required
- Better use of fragmented memory
- Insertion/deletion at middle is easier
- Joining lists easier
- 'Next' operation requires pointer dereference

Contiguous

- Next and Previous are implicit (less storage)
- Can take advantage of locality
- Random access
- 'Next' operation probably faster

Contiguous vs. Linked Data Structures

Linked

- Extra storage required
- Better use of fragmented memory
- Insertion/deletion at middle is easier
- Joining lists easier
- 'Next' operation requires pointer dereference

Contiguous

- Next and Previous are implicit (less storage)
- Can take advantage of locality
- Random access
- 'Next' operation probably faster

Contiguous vs. Linked Data Structures

Linked

- Extra storage required
- Better use of fragmented memory
- Insertion/deletion at middle is easier
- Joining lists easier
- 'Next' operation requires pointer dereference

Contiguous

- Next and Previous are implicit (less storage)
- Can take advantage of locality
- Random access
- 'Next' operation probably faster

Contiguous vs. Linked Data Structures

Linked

- Extra storage required
- Better use of fragmented memory
- Insertion/deletion at middle is easier
- Joining lists easier
- 'Next' operation requires pointer dereference

Contiguous

- Next and Previous are implicit (less storage)
- Can take advantage of locality
- Random access
- 'Next' operation probably faster

Important Data Structures/ADTs

Table 1.1 from Data Structures & Algorithms in Java, by Robert Lafore

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known	Slow search, slow deletion, fixed size
Ordered array	Quicker search than unsorted array	Slow insertion and deletion, fixed size
Stack	Provides last-in, first-out access	Slow access to other items
Queue	Provides first-in, first-out access	Slow access to other items
Linked list	Quick insertion, quick deletion	Slow search
Binary Tree	Quick search, insertion, deletion (if tree remains balanced)	Deletion algorithm is complex
Red-black trees	Quick search, insertion, deletion; tree always balanced	Complex
2-3-4 trees	Quick search, insertion, deletion. Tree always balanced. Similar trees good for disk storage	Complex
Hash table	Very fast access if key known; fast insertion	Slow deletion, access slow if key not known, inefficient memory usage
Heap	Fast insertion, deletion, access to largest item	Slow access to other items
Graph	Models real-world situations	Some algorithms are slow and complex

Arrays

Unordered Array

- We can implement an unordered array in Python using a list

Operations

- `create()`
- `delete()`
- `isEmpty()`
- `length()`
- `find()` k-th element
- `search()` for a given element
- `insert()` an element into the list
- `append()`, `join()`, `copy()`, ...

Ordered Array

- Ordered arrays can also be implemented in Python using a list

Operations

- `create()`
- `delete()`
- `isEmpty()`
- `length()`
- `find()` k-th element
- `search()` for a given element
- `insert()` an element into the list
- `append()`, `join()`, `copy()`, ...

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(lg\, n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(lg\, n)$ ...
- In reality, we can get close, and the price must be paid in complexity

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(lg\, n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(lg\, n)$ ...
- In reality, we can get close, and the price must be paid in complexity

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(\lg n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(\lg n)$ ...
- In reality, we can get close, and the price must be paid in complexity

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(\lg n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(\lg n)$ ...
- In reality, we can get close, and the price must be paid in complexity

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(\lg n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(\lg n)$ ...
- In reality, we can get close, and the price must be paid in complexity

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(\lg n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(\lg n)$ ...
- In reality, we can get close, and the price must be paid in complexity

Why not use arrays for everything?

- Arrays have disadvantages too!
- Unordered Array: insert items quickly ($O(1)$) — but searching is slow, $O(N)$
- Ordered Array: search is quick ($O(\lg n)$) — but insertion is slow, $O(N)$
- Deletion is slow in both cases — half the items on average must be moved to fill the 'hole' ($O(N)$)
- Ideal: data structures that could do everything — insert, delete, search — quickly, perhaps in $O(1)$ time, or at least $O(\lg n)$ ...
- In reality, we can get close, and the price must be paid in complexity

How quick do you need your data structure to be!?

Stacks

Stacks and Queues

- Stacks and queues are more abstract entities than arrays and many other data storage structures
- They're defined primarily by their interface — the permissible operations that can be carried out on them
- The underlying mechanism used to implement them is typically not visible to their users
- The underlying mechanism for a stack can be an array or a linked list

Stacks and Queues

- Stacks and queues are more abstract entities than arrays and many other data storage structures
- They're defined primarily by their interface — the permissible operations that can be carried out on them
- The underlying mechanism used to implement them is typically not visible to their users
- The underlying mechanism for a stack can be an array or a linked list

Stacks and Queues

- Stacks and queues are more abstract entities than arrays and many other data storage structures
- They're defined primarily by their interface — the permissible operations that can be carried out on them
- The underlying mechanism used to implement them is typically not visible to their users
- The underlying mechanism for a stack can be an array or a linked list

Stacks and Queues

- Stacks and queues are more abstract entities than arrays and many other data storage structures
- They're defined primarily by their interface — the permissible operations that can be carried out on them
- The underlying mechanism used to implement them is typically not visible to their users
- The underlying mechanism for a stack can be an array or a linked list

Stack

- Can be implemented using Arrays or Linked Lists

Operations

- `create()`
- `delete()`
- `isEmpty()`
- `push()`
- `pop()`
- `top()`