CE204 Data Structures and Algorithms

David Richerby

University of Essex

Spring term 2021

The list ADT

Operations and behaviour.

- `void create()`, `boolean isEmpty()`, `int length()`.
- `void insert(int index, String s)` insert `s` so it becomes the `index`th item in the list.
- `String get(int index)` returns the `index`th item.
- `void delete(int index)` deletes the `index`th item.

Running time of `insert()`, `get()`, `delete()`is proportional to size of list.

NB: different authors disagree on exactly what the operations should be.

Lists vs arrays

Size:

- List: can grow and shrink as needed.
- Array: size fixed at creation time.

Insert/delete:

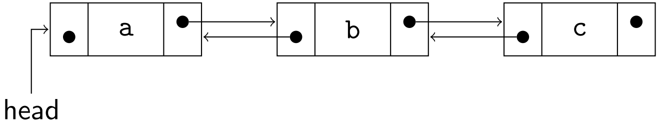
- List: naturally supported.
- Array: requires copying to new array.

Get `i`th item:

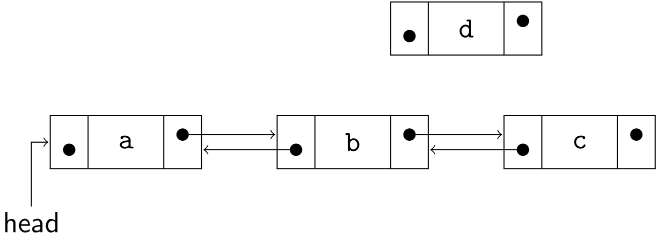
- List: expensive (time $\propto$ length).
- Array: cheap (constant time).

Linked list implementation

Doubly linked list: each item has a reference to the next and previous items.



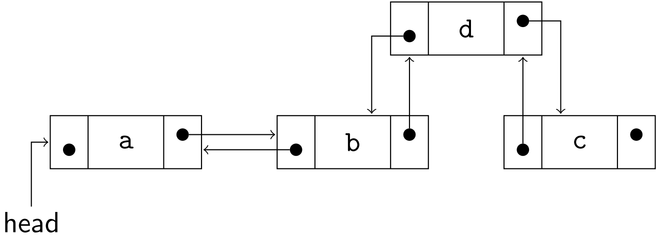
Doubly linked list: each item has a reference to the next and previous items.




```
insert (2, "d");
```

Linked list implementation

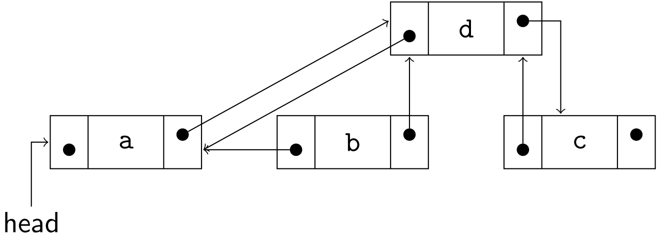
Doubly linked list: each item has a reference to the next and previous items.




```
insert (2, "d");
```

Linked list implementation

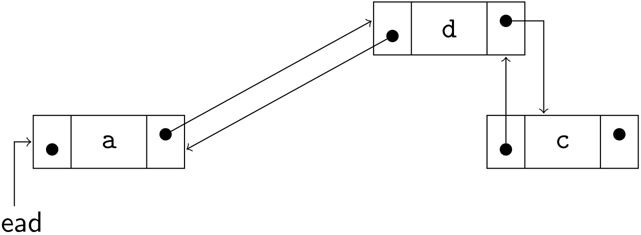
Doubly linked list: each item has a reference to the next and previous items.



`insert (2, "d"); delete (1);`

Linked list implementation

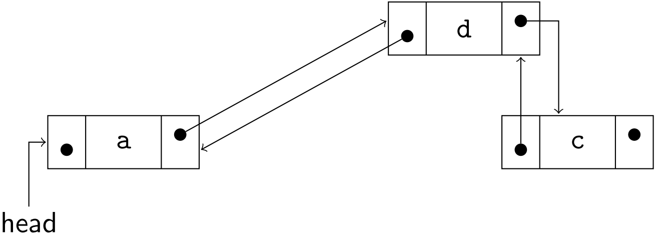
Doubly linked list: each item has a reference to the next and previous items.



`insert (2, "d"); delete (1);`

Linked list implementation

Doubly linked list: each item has a reference to the next and previous items.



`insert (2, "d"); delete (1);`

Singly linked list: each item only links to next.

Java implementation (1)

```
public class DoublyLinkedList {
    private class Item {
        String value;
        Item next;
        Item prev; /* Omitted for singly linked list. */

        Item (String value, Item prev, Item next) {
            this.value = value;
            this.next = next;
            this.prev = prev;
        }
    }

    private Item head = null;
    private Item tail = null;
    private int length = 0;
    ...
```

Java implementation (2)

The following is a useful helper method – returns the `index`th `Item` object in the list.

For internal use only! Caller responsible for ensuring `index` is valid.

```
private Item getItem (int index) {
    Item cur = head;
    for (int i = 0; i < index; i++)
        cur = cur.next;
    return cur;
}
```

NB: `getItem (k)` makes `k-1` steps along the list.

Java implementation (3)

```
public String get (int index) {
    if (index < 0 || index >= length)
        return null;
    else
        return getItem(index).value;
}
```

See source code on Moodle for `insert()` and `delete()`. Similar techniques to stacks and queues.

List iteration (1)

Code like the following is very often needed

```
for (int i = 0; i < list.length(); i++) {
    String s = list.get(i);
    // Do something with s.
}
```

For a list of length n, this makes n calls to `getItem()`.

This makes $0 + 1 + \cdots + n - 1 = \frac{1}{2}n(n-1) \approx n^2$ total steps along the list.

Very expensive: nearly 5 000 steps for a 100-item list.

List iteration (2)

Solution: add a "current" item of the list (a.k.a. "cursor").

```
private Item cur = null;

public void rewind () { cur = null; } /* reset to start */

public String getNext () {
    if (cur == null) cur = head;
    else cur = cur.next;
    return cur == null ? null : cur.value;
}

public boolean hasNext () {
    if (cur == null) return head != null;
    else return cur.next != null;
}
```

List Iteration (3)

Now we can efficiently iterate through our list:

```
list.rewind ();
while (list.hasNext ()) {
    String s = list.getNext ();
    // Do something with s.
}
```

This only takes $n - 1$ steps along the list.

Can similarly implement `hasPrev()` and `getPrev()` to go backwards (not in a singly linked list).

Also `addToHead ()` and `addToTail ()` as efficient common cases for extending the list.

Java standard libraries

Two implementations of lists: `ArrayList` and `LinkedList`.

`ArrayList` stores entries in an array

- get is fast (constant time);
- insertion and deletion are slow (time $\propto$ length);
- append is fast (constant time) on average, like array stack.

`LinkedList` is a doubly linked list.

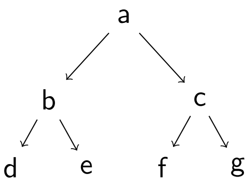
`ListIterator` interface and `List.listIterator()` method define iterators. Also allow insertion, deletion, etc. at current position.

Lists: summary

- Like growable arrays with insertion and deletion.
- Random access typically expensive.
- Efficient iteration possible.
- Different implementations suited to different purposes.

Binary Trees

Binary trees



- Each entry (a, b, c, . . .) is a **node**.
- Every node except one has a **parent** above it.
- The node with no parent (a) is the **root**.
- Each node has a **left child** or a **right child** or neither or both.
- A node with no children is a **leaf**.
- The **subtree rooted at a node** is the tree consisting of that node and all its descendants.

The (traditional) binary tree ADT

Operations and behaviour.

- `void create (String s)` creates a one-node tree that holds `s`.
- `void join (String s, Tree left, Tree right)` creates a tree with `s` at the root and the specified subtrees.
- `boolean isLeaf()` checks if the tree has just one node.
- `void Tree leftChild()` and `void Tree rightChild()` return the subtrees rooted at the children of the root.
- `String value()` returns the string stored at the root.

Running time of all operations is independent of the size of the tree.

ADT critique

The given operations aren't actually very useful – they require trees to be built from leaves upwards, e.g.,

```
Tree d = new Tree ("d");
Tree e = new Tree ("e");
Tree c = new Tree ("c", d, e);
...
Tree a = new Tree ("a", b, c);
```

Practical implementations typically add operations.

Java implementation (1)

```
public class BinaryTree {
    private String value;
    private BinaryTree left;
    private BinaryTree right;

    BinaryTree (String value) { this (value, null, null); }

    BinaryTree (String value, BinaryTree left, BinaryTree
        right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
    ...
```

Java implementation (2)

```
    boolean isLeaf () { return left == null && right ==
        null; }

    BinaryTree leftChild () { return left; }
    BinaryTree rightChild () { return right; }
    String value () { return value; }
}
```

Code for binary trees often uses recursion: much easier than trying to walk up and down the tree with a `cur` reference.

```
boolean contains (String s) {
    return (value != null && value.equals (s))
            || (left != null && left.contains (s))
            || (right != null && right.contains (s));
}
```

Traversing binary trees

Traversing a tree refers to systematically walking through its nodes and processing them.

Three main methods:

- pre-order: process a node, then recurse on the subtrees.
- post-order: recurse on the subtrees, then process the node.
- in-order: recurse on the left subtree, process the node, then recurse on the right subtree.

`contains()` on the previous slide is essentially a pre-order traversal that stops early if it finds `s`.

Pre-order traversal

```
void preOrder () {
    System.out.println(value);
    if (left != null) left.preOrder();
    if (right != null) right.preOrder ();
}
```

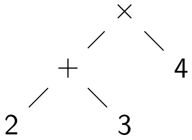
Uses:

- searching the tree;
- printing the tree, viewing each node as a heading with two subsections (e.g., “Noah begat Shem and Ham, and Shem begat Elam and Arpachshad, and Arpachshad begat...”).

Post-order traversal

```
void postOrder () {
    if (left != null) left.postOrder();
    if (right != null) right.postOrder ();
    System.out.println(value);
}
```

Used to evaluate arithmetic expressions stored as trees.

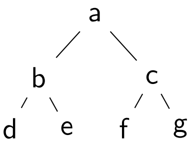


To evaluate a non-leaf node, evaluate the children and combine with the arithmetic operator at the node. $(2 + 3) \times 4 = 20$.

In-order traversal

```
void inOrder () {
    if (left != null) left.inOrder();
    System.out.println(value);
    if (right != null) right.inOrder ();
}
```

Used to print the contents of the tree “left-to-right”.



In-order: d, b, e, a, f, c, g

Non-binary trees

In general, nodes in trees may have any number of children.

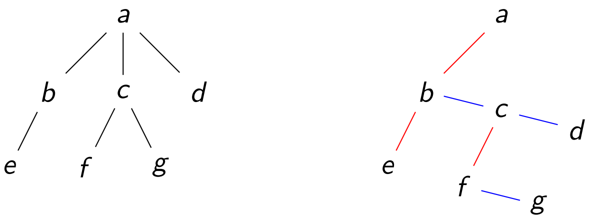
- If maximum number of children is known, each node could have an array of children. Wastes space if most nodes have fewer children. Are you *sure* it's the max? ("Nobody has more than ten children, right?")
- If number of children known is when node is created, and can't change, can make array of the right size.
- Usually better to use a list of children.

Left-child, right-sibling

To avoid needing separate tree and list classes, we can store child lists *implicitly* in a binary tree.

Repurpose the left and right child references:

- `left` points to the node's first child (in red, below);
- `right` points to the node's next sibling (in blue).



Uses of (binary) trees

- Binary search trees and priority queues are efficient ways of storing and searching sorted data (next lecture).
- 4-ary trees used to partition space in computer graphics.
- Syntax trees in compilers.
- Hierarchical data.