CS 234: Data Types and Structures

Naomi Nishimura

Module 10

Date of this version: November 26, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

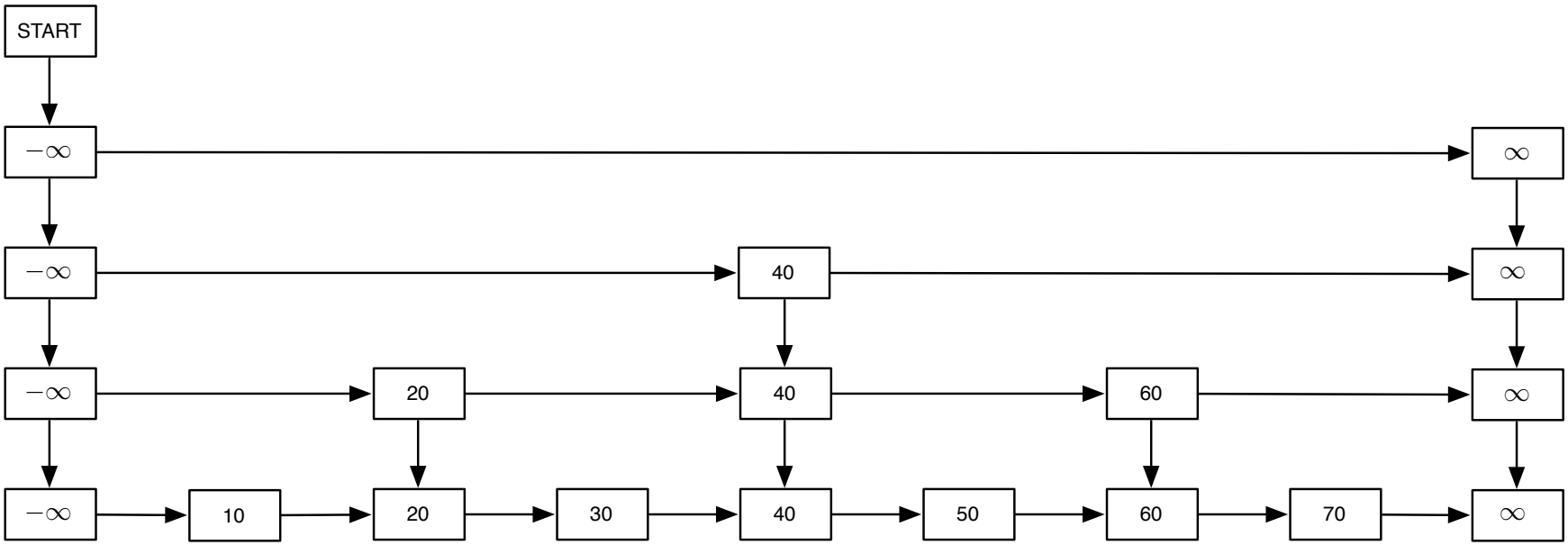
Directions for further inquiry

- New data structures for common ADTs
- New ways of using ADTs together
- New ADTs and data structures for special kinds of data
- New criteria for data structures

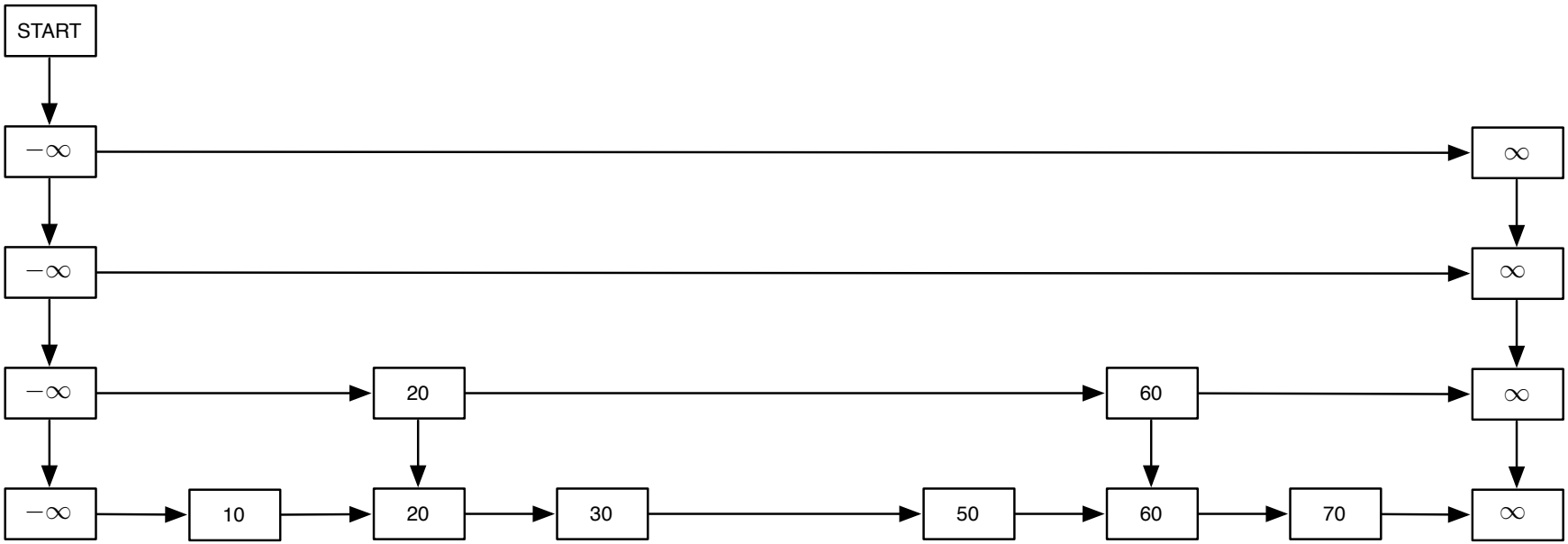
New data structures for ADT Dictionary

Skip list:

- Combines strong points of linked (easy to modify) and contiguous (binary search)
- Linked nodes of different “heights”



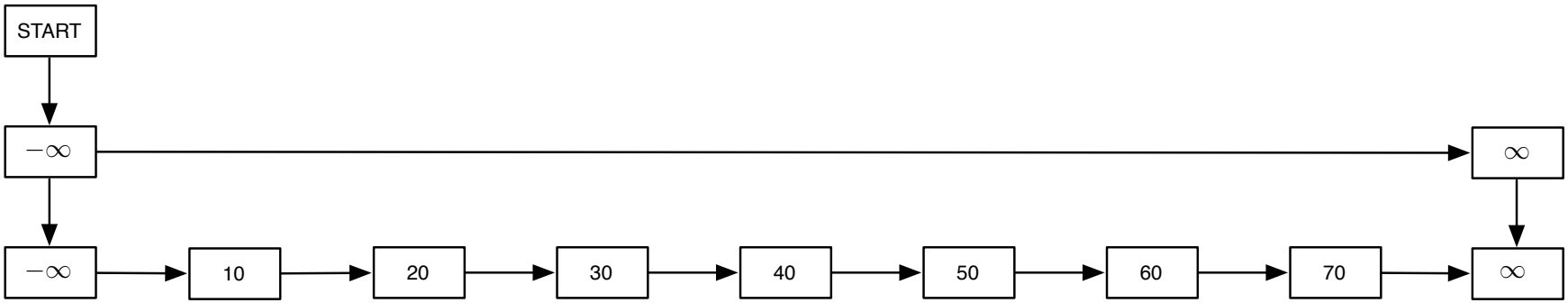
Modifying a skip list



Using randomization in a skip list

Use randomization to keep nice distribution of heights even as changed by addition and deletion.

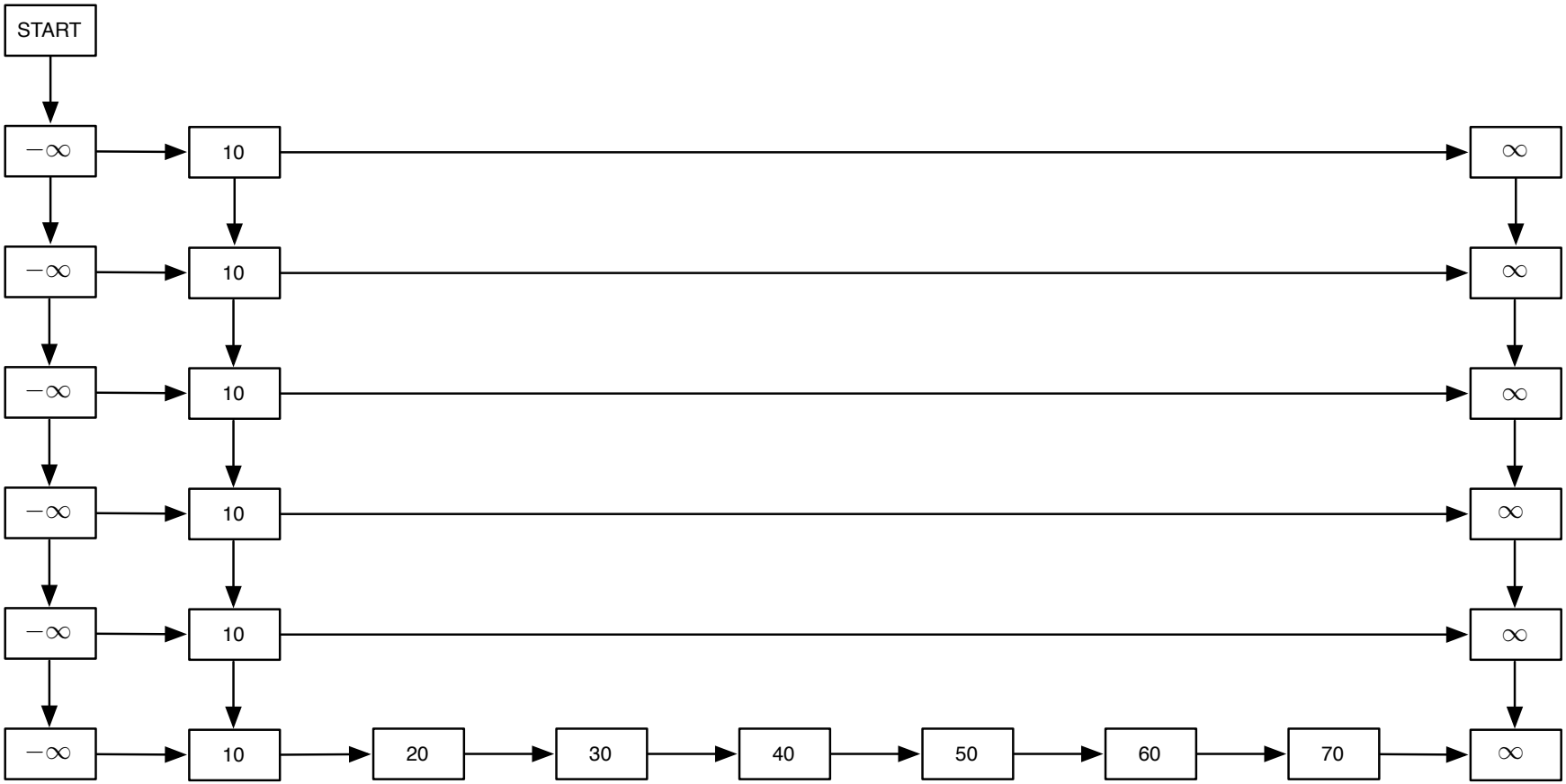
In one bad case, all data items have the same height.



The cost of search is now linear in the number of data items in the worst case.

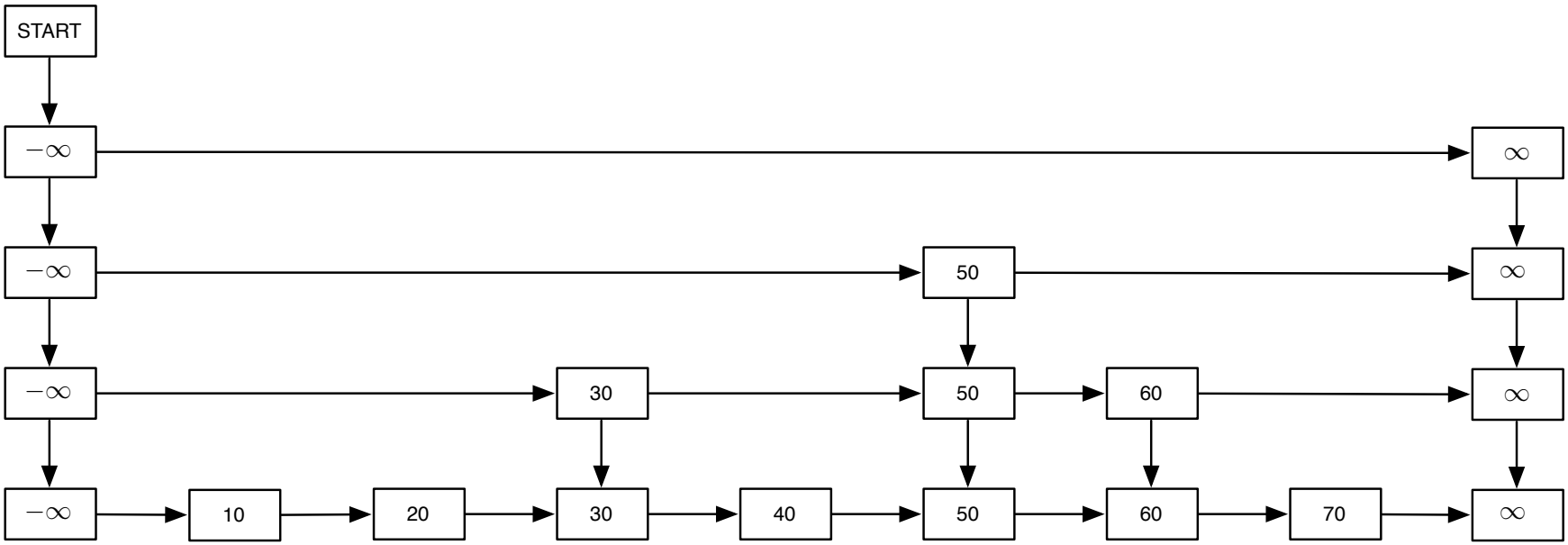
Bad cases for skip lists

In another bad case, one or more items may have a very large height.



A possible skip list

Key idea: Good expected behaviour.



New ways of using ADTs together

Minimum spanning tree: Given a weighted graph, choose edges to form a tree including all vertices in the graph such that the sum of the weights is minimized.

Total order: Given a list of tasks and information about which should be done before which, determine an ordering.

More problems solved using ADTs

Connectivity: Given a graph, determine the **connected components** (sets of vertices such that there is path between each pair of vertices in the set).

Single-source shortest paths: Given a weighted graph and a source vertex, determine the minimum costs of paths from the source to each other vertex.

ADT Disjoint Set

Preconditions: For all *D* is a set of disjoint sets and *Data* any data item; for *FIND_REP*, *Data* is in *D*; for *ADD_SET*, *Data* is not in *D*; for *UNION* *One* and *Two* are data items in *D*.

Postconditions: Mutation by *ADD_SET* (creates new set containing only *Data*) and *UNION* (combines sets containing data items *One* and *Two*).

Name	Returns
CREATE()	a new empty set of disjoint sets
FIND_REP(D, Data)	data item that serves as representative of set containing *Data*
ADD_SET(D, Data)	
UNION(D, One, Two)	

Extensions to ADT Priority Queue

- Allow key values to change. (ADT Mergeable Heap)
- Allow two structures to be joined into one. (ADT Meldable Heap)

Special data: Multi-dimensional data

Data:

- Data items with d aspects or coordinates

Applications:

- Data with multiple attributes (e.g. products with different dimensions, weights, prices)
- Spatial data - points in d-dimensional space (e.g. graphics, computational geometry, computer-aided design, geographic data)

Types of problems solved on multi-dimensional data

- Range queries: Find all data items in a certain specified range
- Nearest-neighbour queries: Find the data item closest to a specific query point
- Partial key search: Find a data item specified by only certain dimensions

ADT Range

For simplicity, we start with search on a range in one dimension of data.

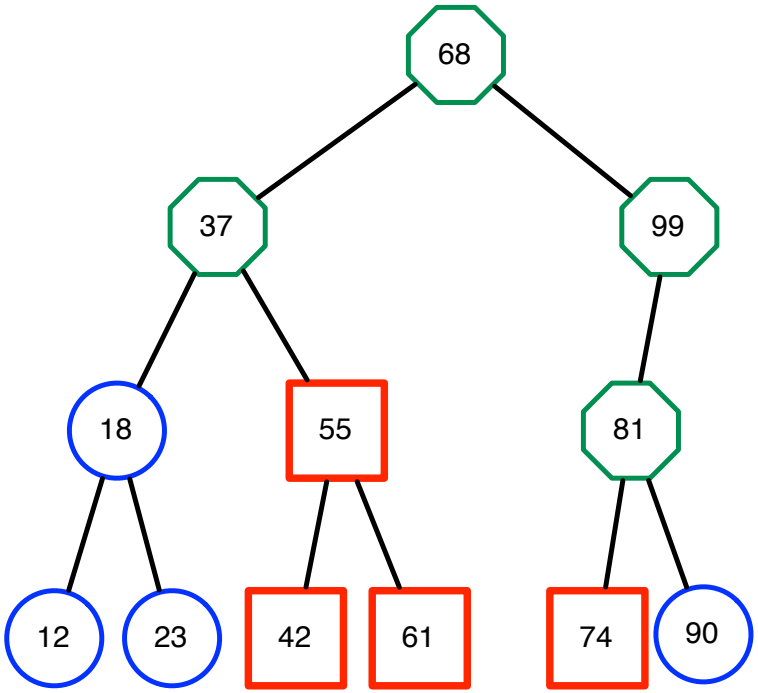
Preconditions: For all *R* is a range and *Data* any data item; for *RANGE*, *Low* and *High* are values; for *ADD Data* is not in *R*; for *DELETE Data* is in *R*.

Postconditions: Mutation by *ADD* (adds item) and *DELETE* (deletes item).

Name	Returns
CREATE()	a new empty range
RANGE(R, Low, High)	all data items in *R* with values in the range from *Low* to *High*
ADD(R, Data)	
DELETE(R, Data)	

Implementing ADT Range using a BST

Range from 30 to 80



- **Inside node**: It and its descendants are all in the range
- **Outside node**: It and its descendants are all outside the range
- **Boundary node**: It has descendants both in and out of the range

Algorithm sketch

Run search recursively:

- On the left subtree only if the current value is greater than the entire range.
- On the right subtree only if the current value is smaller than the entire range.
- On the left, include the current value, and on the right, otherwise.

Data structure: Quadtree

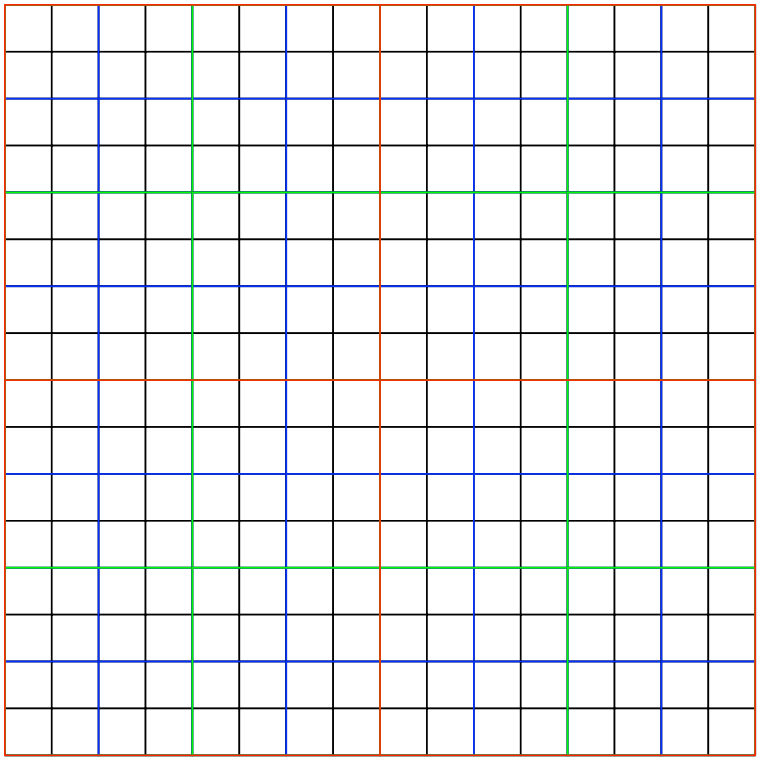
For two-dimensional data:

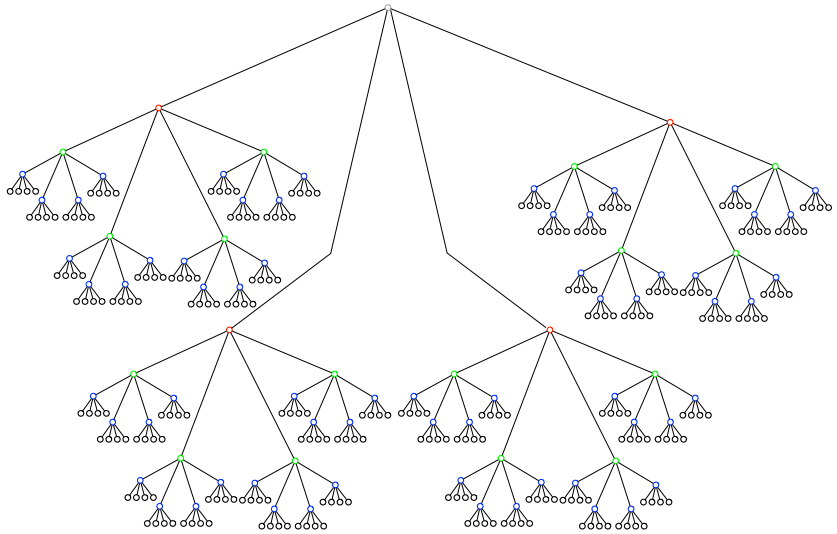
- Represent a square region as a tree
- Root represents entire region
- Four children for four quadrants
- Leaves store points

Variants and uses:

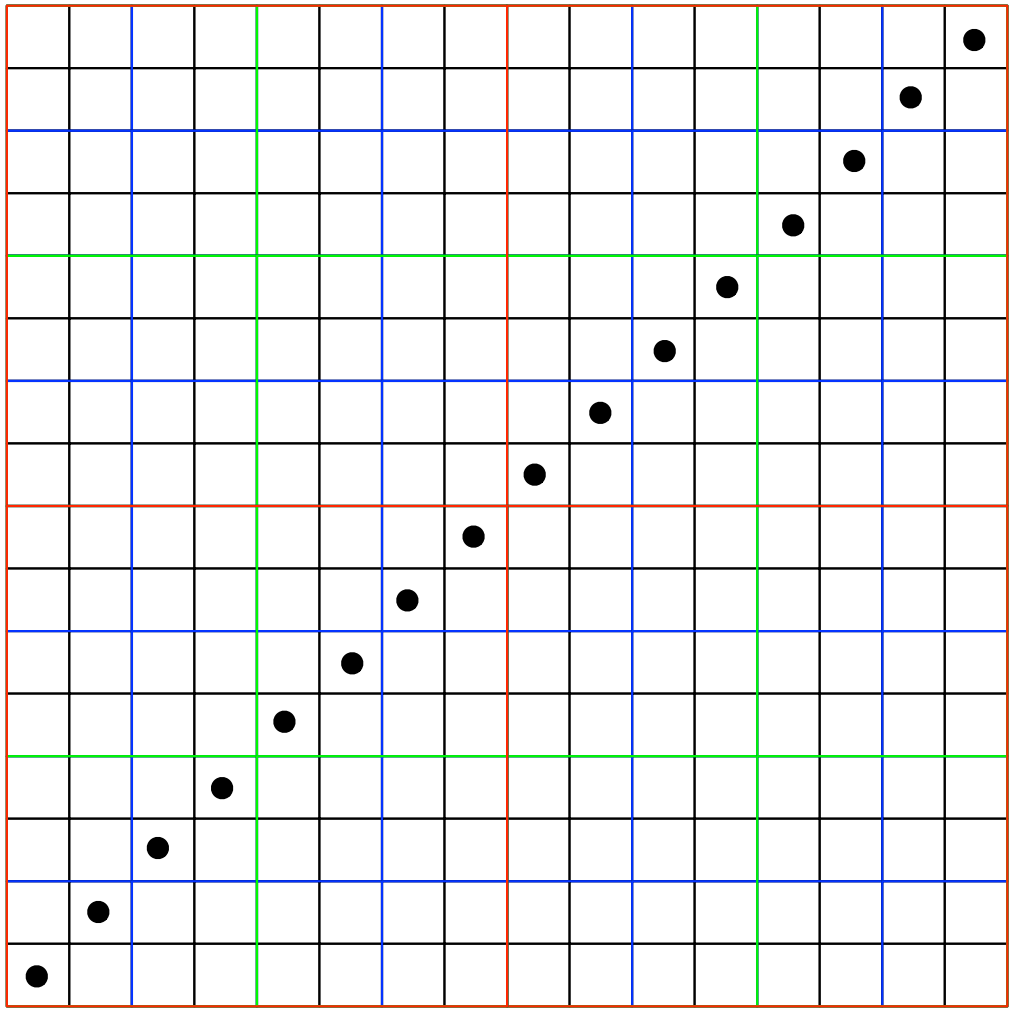
- Good for compression of image data
- Can lead to high costs for “imbalanced” point data
- Octtrees for 3-dimensional data
- Poor generalization for higher dimensions

Breaking an area into a grid





Problematic data for a quadtree



Data structure: k-d tree

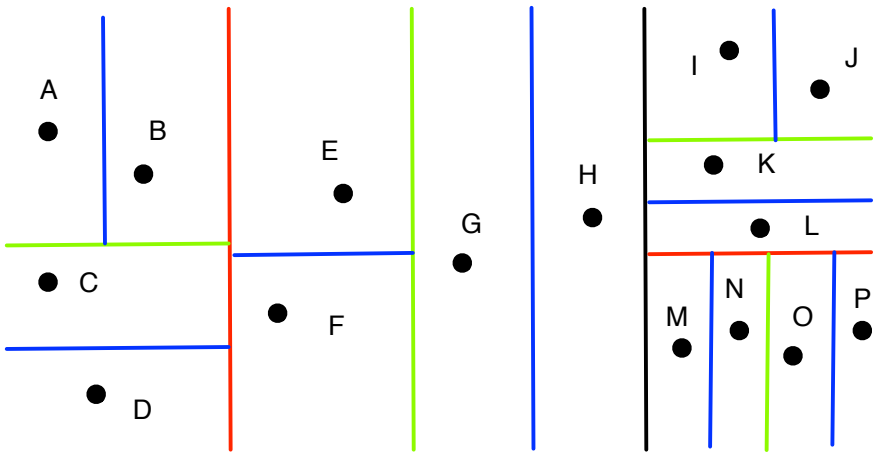
For k-dimensional data:

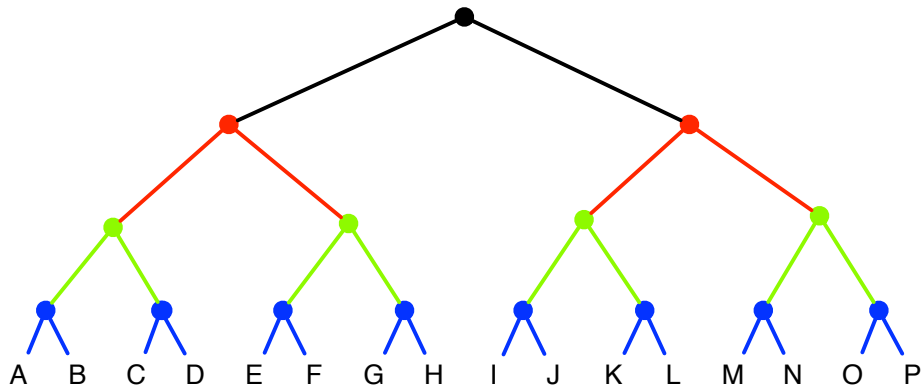
- Represent a region as a binary tree
- Nodes represent points
- Splits depend on distributions of points in a region

Creating a k-d tree:

- Find dimension which represents biggest range in values
- Choose median point on that dimension for new node
- Repeat to logarithmic depth

k-d tree example





Special data: Strings

Data:

- Data items form sequences of symbols chosen from a fixed-size set

Applications:

- Textual information
- Bioinformatics

Types of problems:

- Coding and decoding
- Text compression

Data structure: Trie

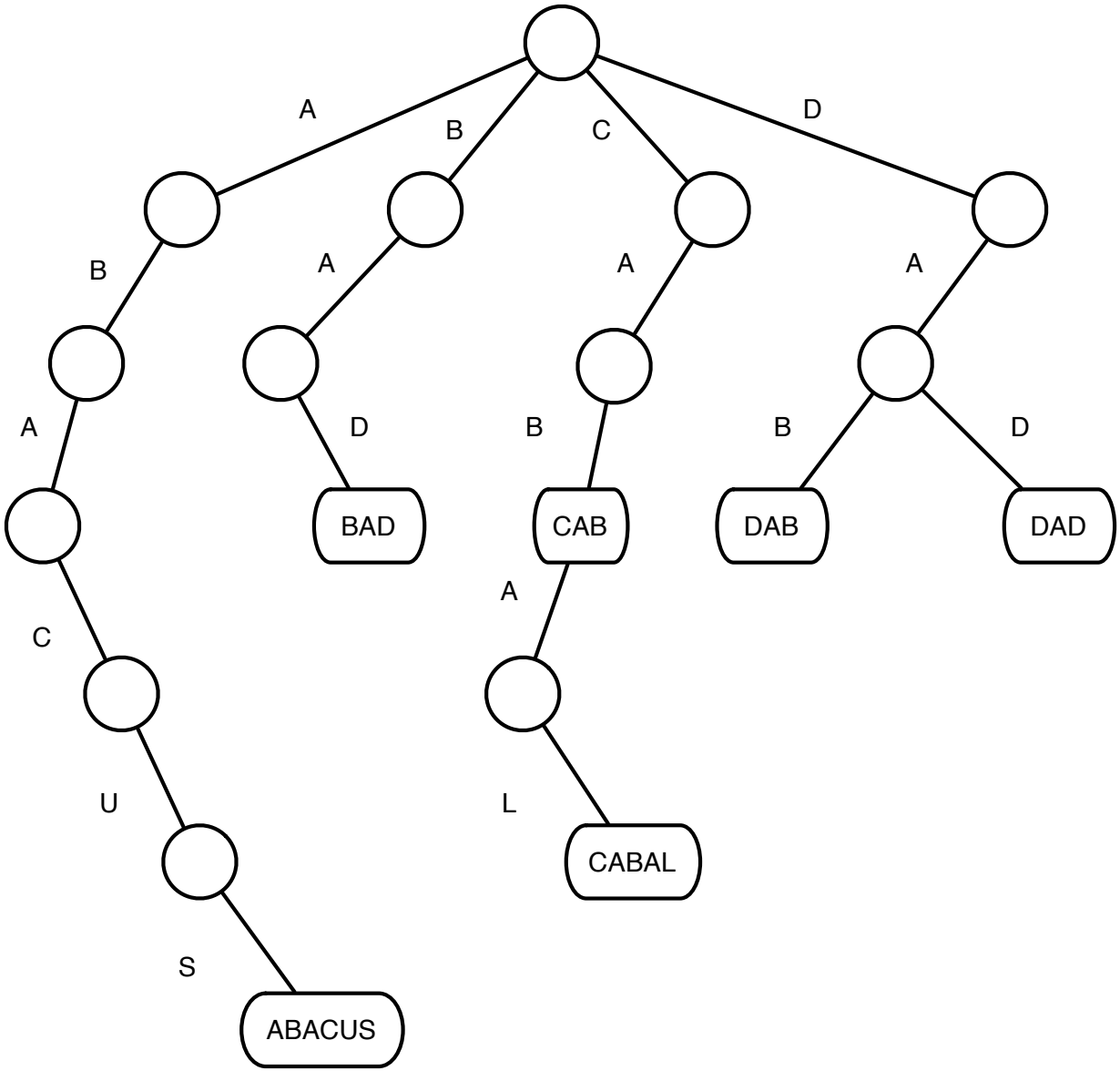
A **trie** stores strings as nodes in an ordered tree, where each node can have at most one child for each possible symbol, and the node storing a string can be found from the root by processing each symbol in turn, taking the child with that symbol at each step.

The term trie was originally pronounced “tree” for “reTRIEval” and now is also pronounced “try”.

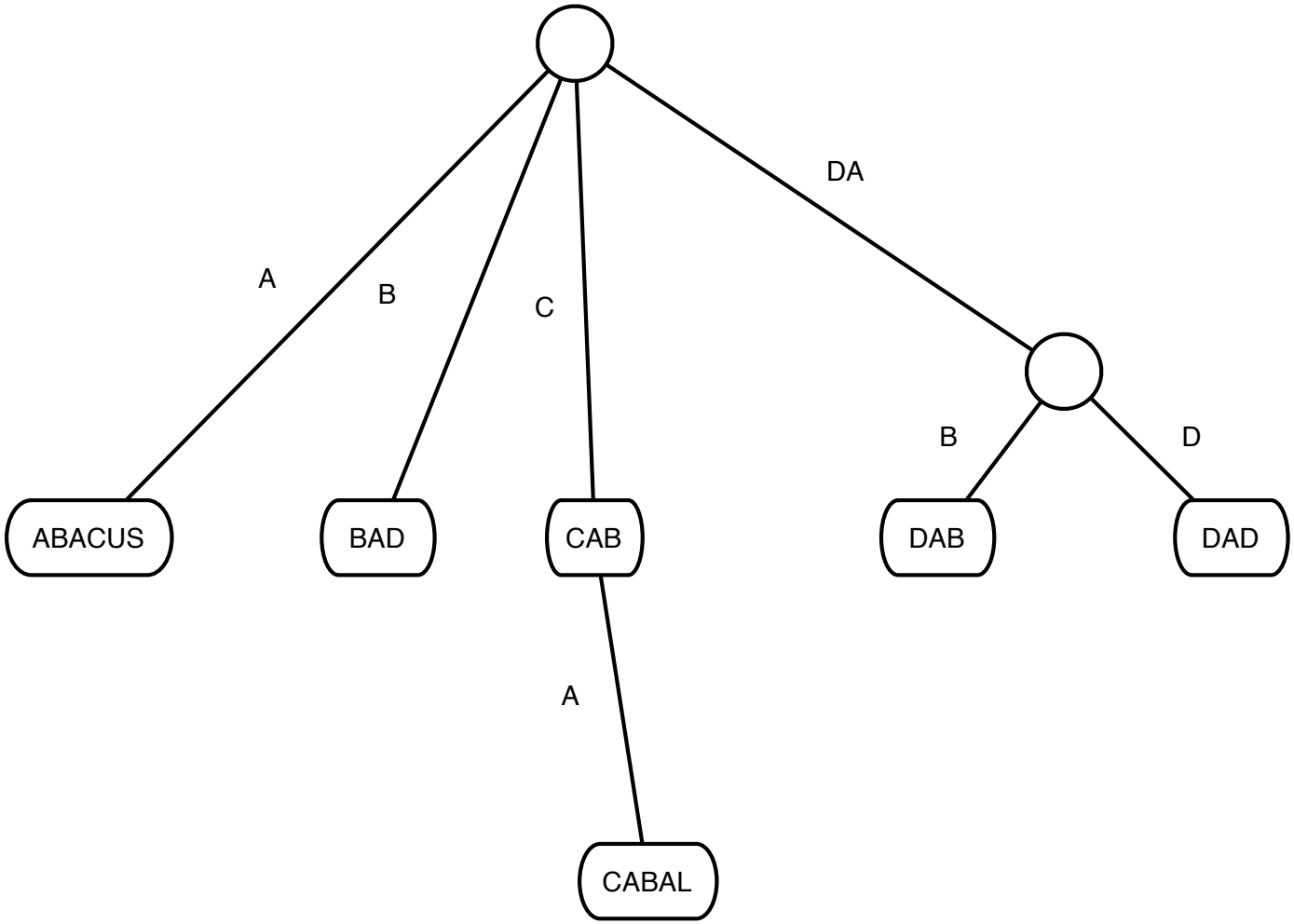
Variant: **Compressed trie**.

A **suffix tree** is a trie used to store all the suffixes of a string, and can be used for pattern matching.

Trie illustration



Compressed trie illustration



New criteria and analysis for data structures

Kinetic data structures capture the idea of motion.

Persistent data structures allow multiple versions to be stored simultaneously.

Succinct data structures use as little space as possible.

Amortized analysis determines the worst-case cost of a sequence of operations; an expensive operation might only occur after many cheap operations.

A little more information about memory

The **memory hierarchy** consists of different types of memory, where the size of each type increases as the access speed decreases:

- registers
- cache (multiple levels)
- main memory
- secondary storage
- tertiary storage

All you need to know for this course is that there is a hierarchy.

Simplify to two levels:

- **Main memory**
- **External memory** (e.g. disks, cloud)

Impact on data structure design

Moving data between levels:

- Processing takes place in main memory.
- Data to be processed is moved into main memory as it is needed, and moved out to make room for other data to be processed.
- Data is moved between levels in fixed-sized chunks (**pages**).
- There are various **paging algorithms** used to determine which page(s) to move out when a new page is moved in.

Bottom line:

- For small amounts of data, only main memory needs to be considered.
- For large amounts of data, take into account page size.
- The number of accesses to external memory may dominate the cost of an algorithm.

Data structure: B-tree

B-tree of order d:

- Generalization of a (2,3) tree
- Choose d so that d data items fill a page
- The root has at most d children
- Other nodes have at least $\lceil d/2 \rceil$ and at most d children

Operations:

- $d+1$ children split into nodes with $\lceil (d+1)/2 \rceil$ and $\lfloor (d+1)/2 \rfloor$ items
- $\lceil d/2 \rceil - 1$ children joined to become a node of size at most d

Variants:

- **B+-tree**: All data appears in the leaves, "threaded" (links between leaves); increase space use by being judicious about splits
- A B+-tree of order 256 holding 4 million keys uses at most 3 disk accesses

Future directions

To use what you have learned:

- Planning - get practice solving problems from lots of application areas
- Coding - code some of the many variants discussed but not coded in class

To learn more:

- CS 230 on hardware and software
- CS 231 on algorithms
- CS 338 on managing large amounts of data
- Courses in application areas of computer science

Module summary

Topics covered:

- Directions for further inquiry
- Data structure: Skip list
- New ways of using ADTs together
- ADT Disjoint Set
- Extensions to ADT Priority Queue
- Special data: Multi-dimensional data
- ADT Range
- Data structure: Quadtree
- Data structure: $k-d$ tree
- Special data: Strings
- Data structure: Trie
- New criteria and analysis for data structures
- Memory hierarchy
- Data structure: B-tree
- Future directions