CSCI203

Algorithms and Data Structures

Quadtrees and Fast Search

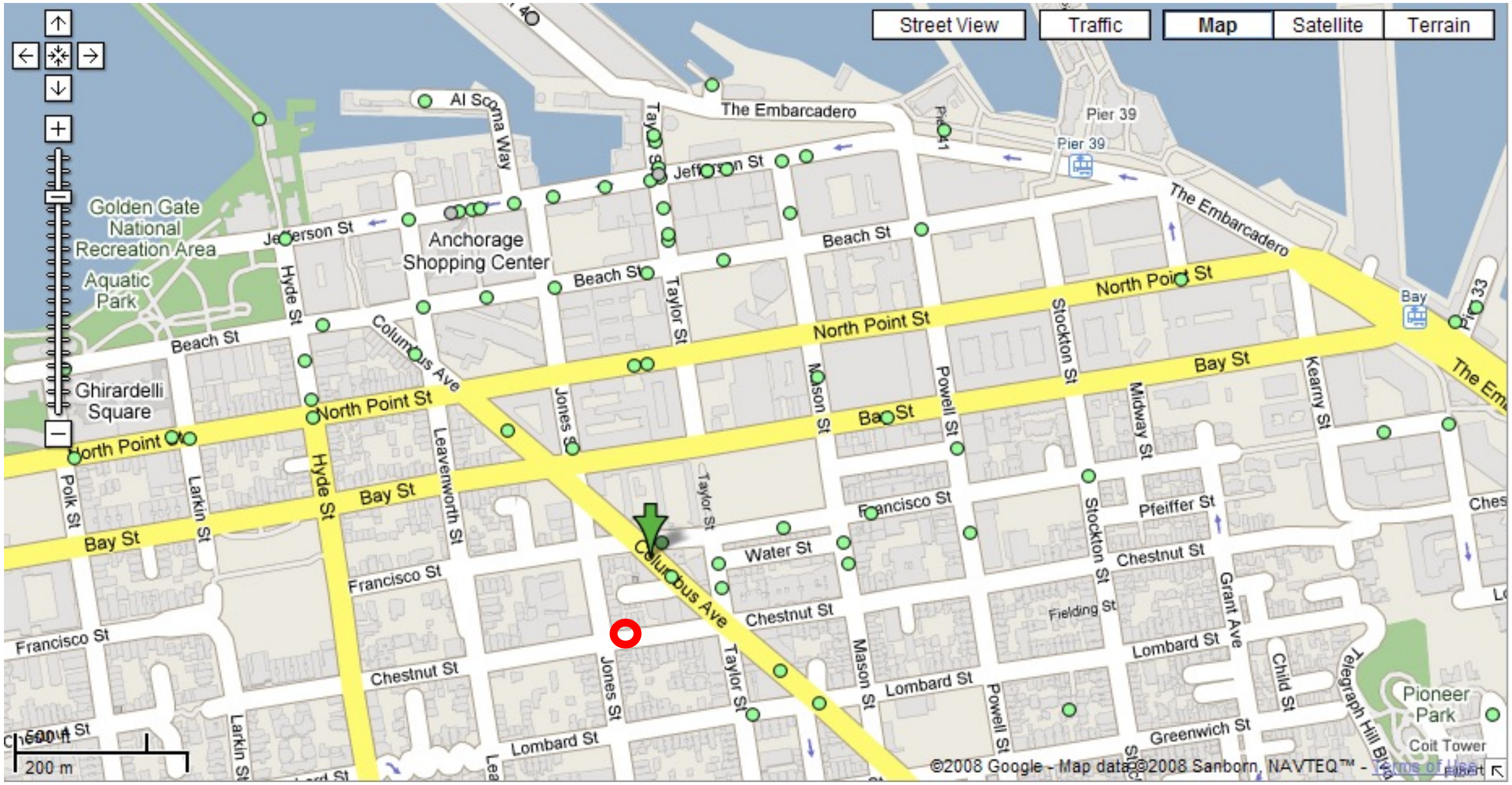
Lecturer: Dr. Zuoxia Yu
Room 3.116
Email: zyu@uow.edu.au

Quadtrees

- Applications of Geometric /Spatial Data Structures
  - Computer graphics, games. Movies
    - Mesh generation
    - Collision detection in 2D
  - Computer vision, CAD, street maps (Google maps / Google Earth)
    - Image representation and processing
  - Human-Computer interface design
  - Virtual reality
  - Visualization (graphing complex functions)

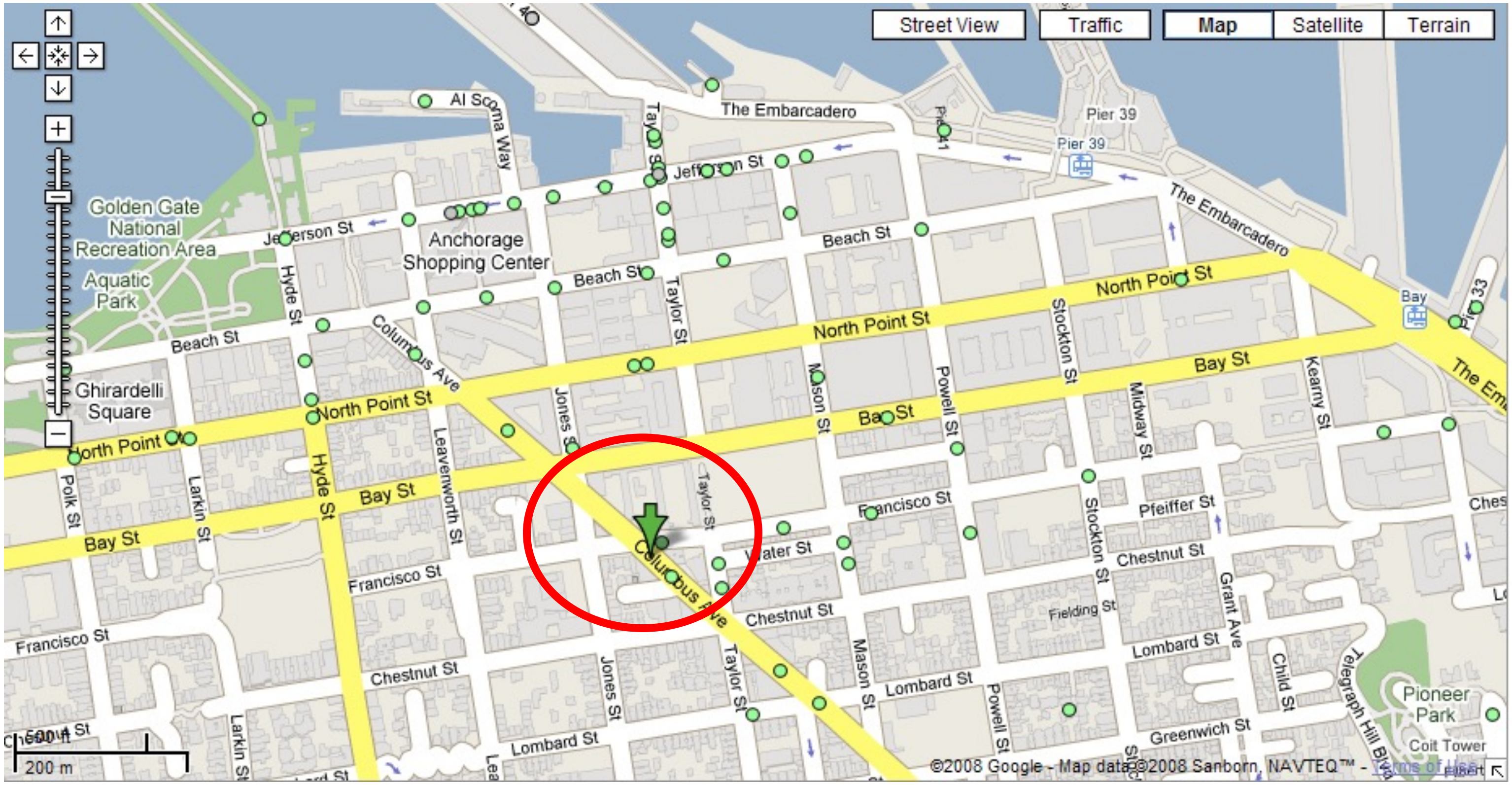
Quadtree in Actions

- What is the closest restaurant to my hotel?



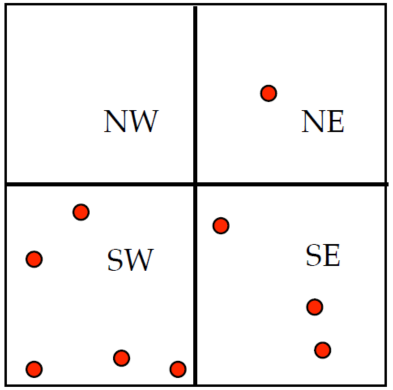
Quadtree in Actions…

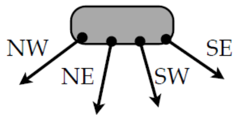
- Find the 4 closest restaurants to my hotel



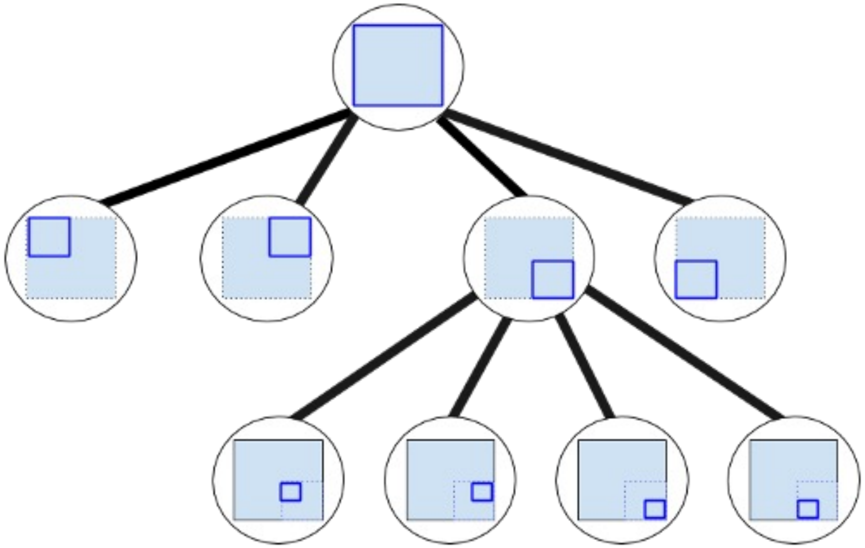
Quadtrees

- Most often used to partition a 2D space



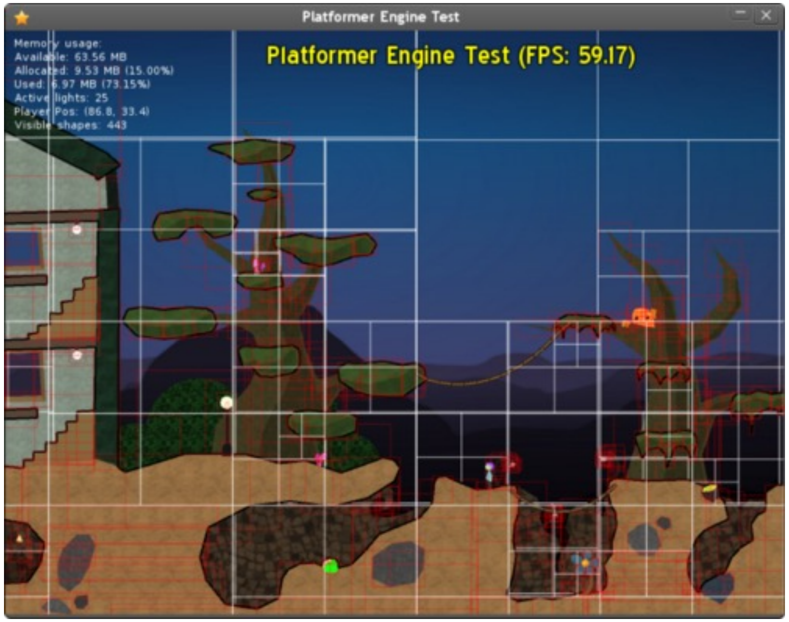


- Each internal node (including the root) has *exactly* four children
  - 4-way tree



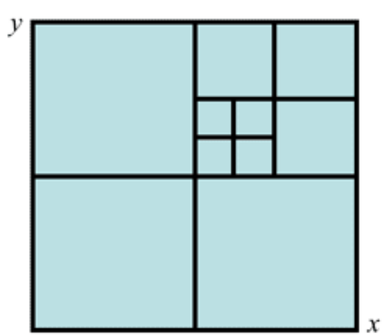
Quadtrees…

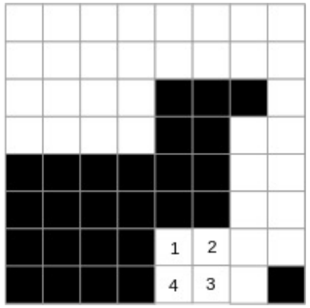
- Data associated with leaf node represents "interesting" information



Types of Quadtrees

- Region Quadtrees
  - recursive subdivisions into squares
  - data stored in a leaf node is information about the space of the cell it represents.
  - often used in image processing.





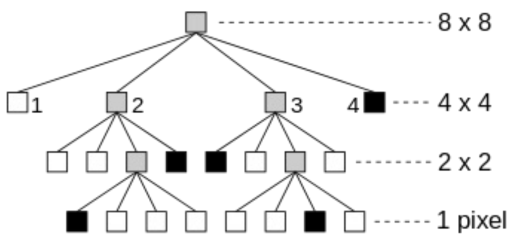
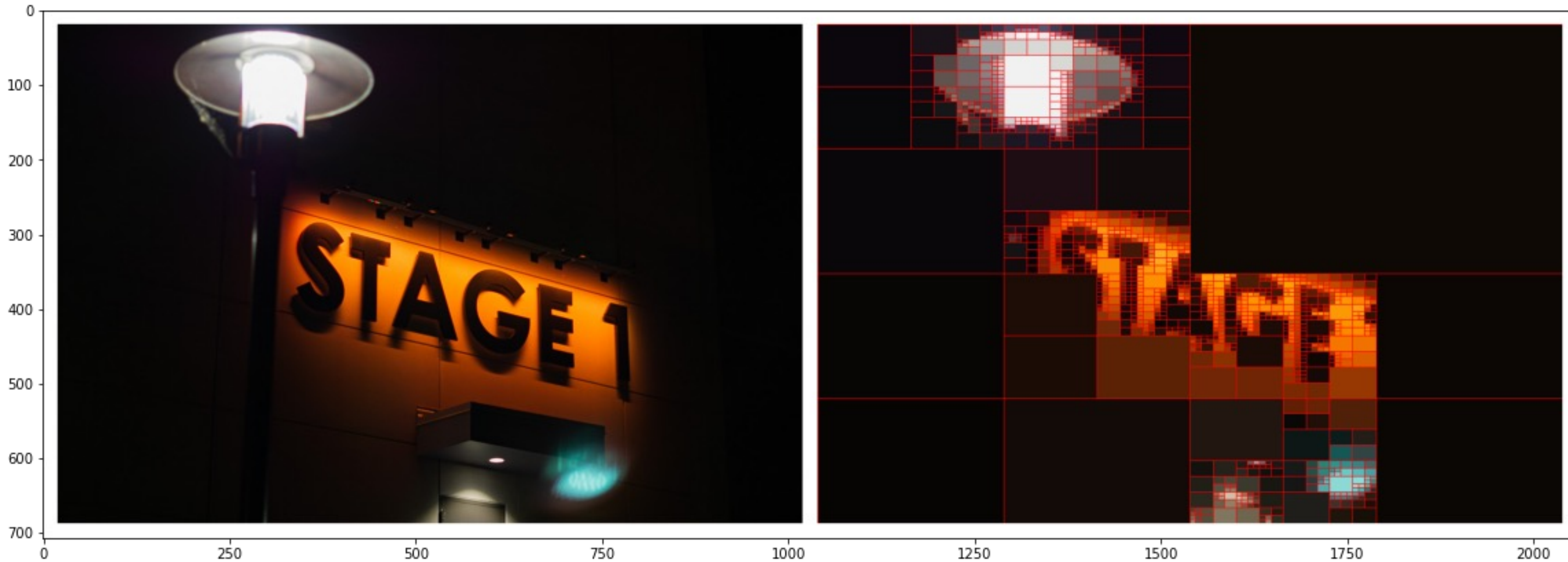
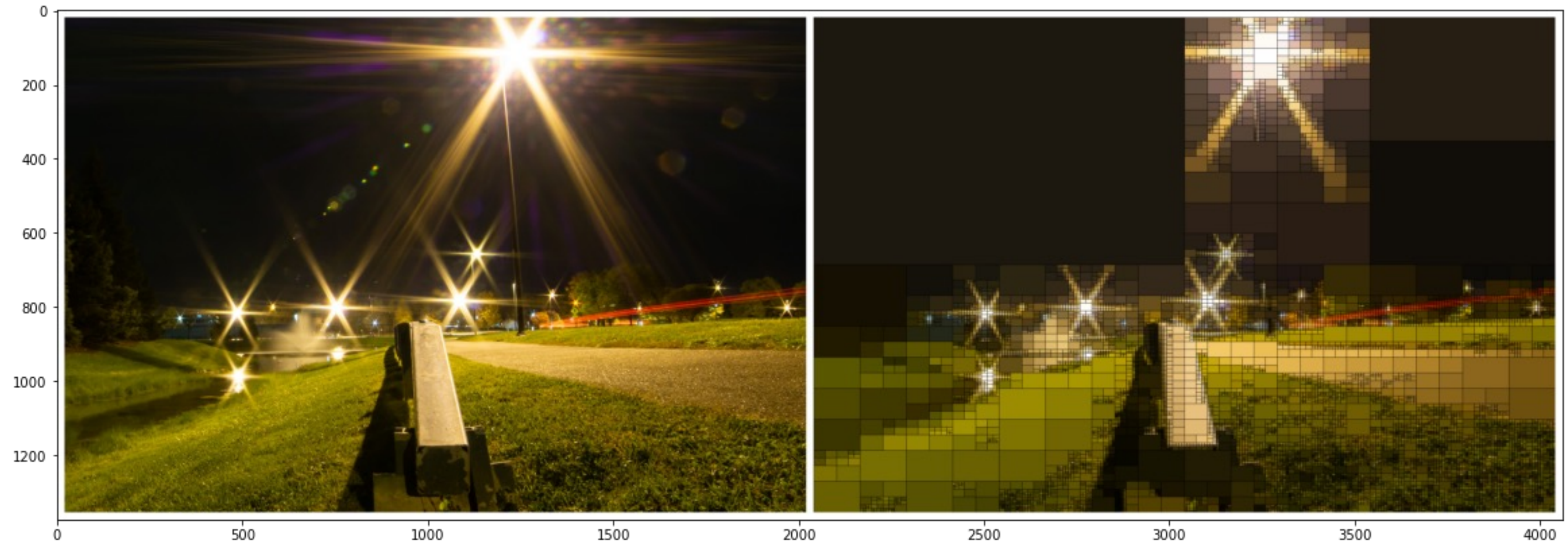


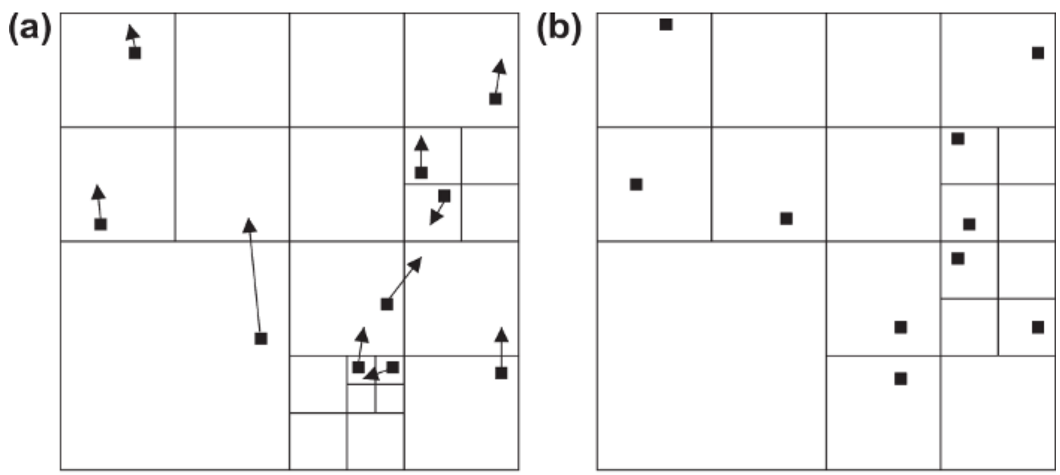
image segmentation


STAGE 1
0
100
200
300
400
500
600
700
0
250
500
750
1000
1250
1500
1750
2000

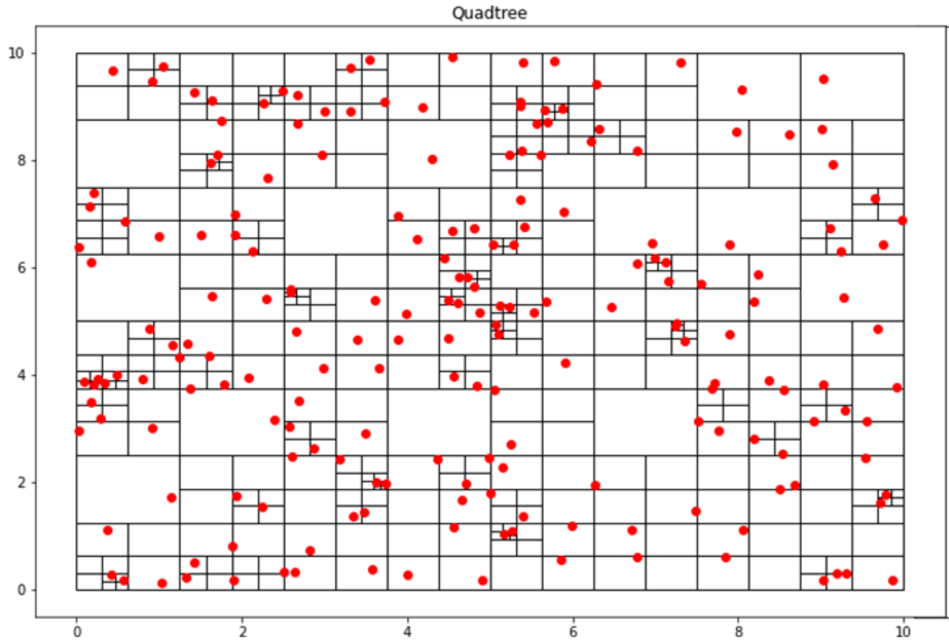

0
200
400
600
800
1000
1200
0
500
1000
1500
2000
2500
3000
3500
4000

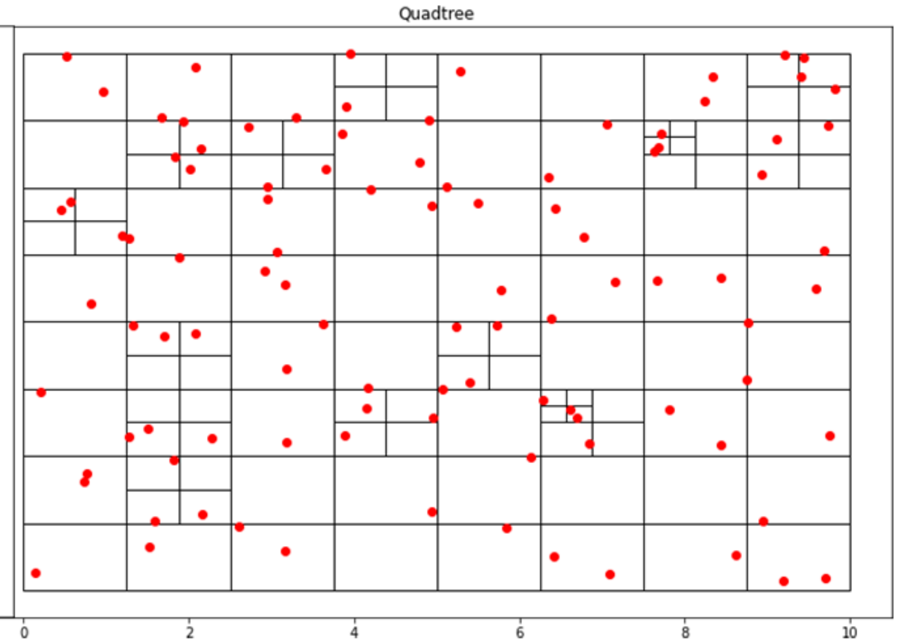
Common Types of Quadtrees…

- Point and Point-Region Quadtrees
  - Similar to region quadtrees
  - PR quadtrees store a list of points that exist within the cell of a leaf, rather than values applied to an entire area of the cell of a leaf in a region quadtree



particle simulation





Each cell has one point

Each cell has one or two points

Three types of nodes are used in PR quadtree:

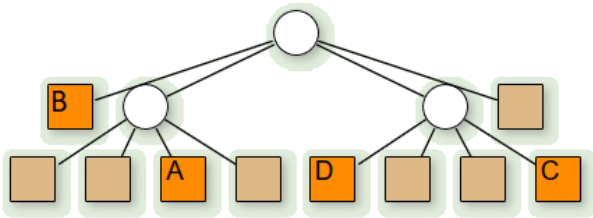
- **Point node**: Used to represent of a point. Is always a leaf node.
- **Empty node**: Used as a leaf node to represent that no point exists in the region it represent.
- **Region node**: This is always an internal node. It is used to represent a region.
  A region node always have 4 children nodes that can either be a point node or empty node.

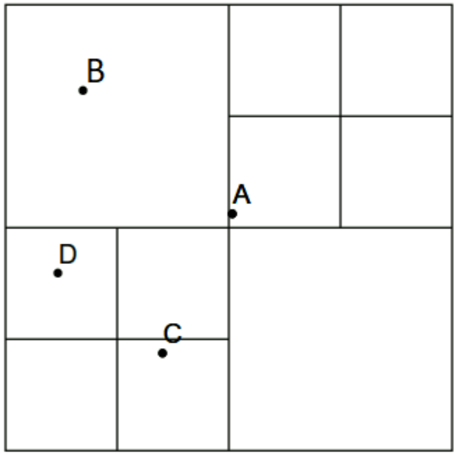
PR Quadtrees - Operations

- Construction from a 2D area
  1. Divide the current two dimensional space into four regions
  2. If a region contains one or more points in it,
     a. create a child object, storing in it the two dimensional space of the region
  3. If a region does not contain any points,
     a. do not create a child for it
  4. Continue to perform recursion for each of the children

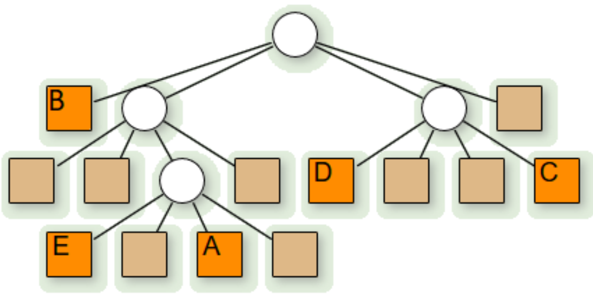
PR Quadtrees - Operations…

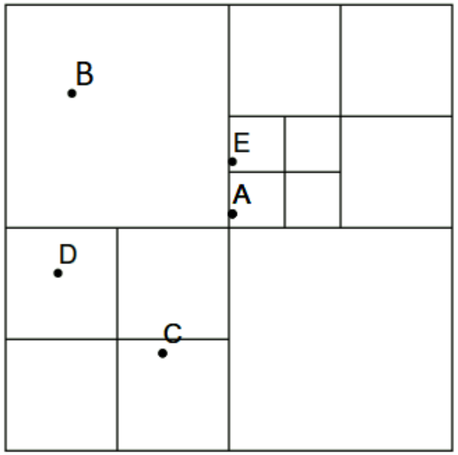
- Insertion
  1. Start with root node as current node.
  2. If the given point is not in boundary represented by current node, stop insertion with error.
  3. Determine the appropriate child node to store the point.
  4. If the child node is empty node, replace it with a point node representing the point. Stop insertion.
  5. If the child node is a point node, replace it with a region node. Call insert for the point that just got replaced. Set current node as the newly formed region node.
  6. If selected child node is a region node, set the child node as current node. Recursively call insertion.





Insert E



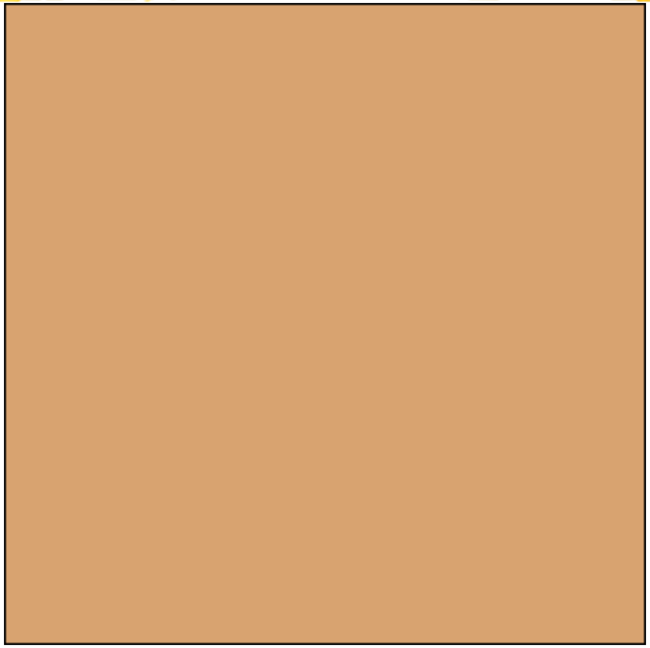


PR Quadtrees - Operations…

- Search
  1. Start with root node as current node.
  2. If the given point is not in boundary represented by current node, stop search with error.
  3. Determine the appropriate child node to search the point.
  4. If the child node is empty node, return FALSE.
  5. If the child node is a point node and it matches the given point return TRUE, otherwise return FALSE.
  6. If the child node is a region node, set current node as the child region node. Recursively call search.

Quadtrees - Complexity

- Time complexity:
  - Find: $O(\log N)$
  - Insert: $O(\log N)$
  - Search: $O(\log N)$



fogleman

- Space complexity:
  - $O(k \log N)$
  - Where k is count of points in the space and space is of dimension $N \; x \; M$, $N \;>=\; M$.

Faster Searching

- Say we have a set of data consisting of pairs:
  - E.g. Name and Telephone Number.
- How do we find the number associated with a given name?
  - What is the best data structure to use?
  - Assume that there are n pairs of data.
- Linear list:
  - Look at every entry.
  - $\Theta(n)$.
- BST:
  - Traverse the tree from the root.
  - $\Theta(\log n)$, $O(n)$ worst case…
  - …so use a balanced tree.

Even faster ?

- Can we do better than $\Theta(\log n)$?
- How about $\Theta(1)$?
  - Constant time searching.
- To do this we use a dictionary:
  - Map;
  - Hash table.
- This is a data structure that allows you to determine:
  - Whether a key is present;
  - If a key is present what its associated data is.

Constant Time Search $\Theta(1)$

- Use a Dictionary:
  - Map + Hash table.
- This is a data structure that allows you to determine:
  - Whether a key is present;
  - If a key is present what its associated data is.

Operations on Dictionaries

- Given a dictionary D, with contents consisting of pairs of the form $< key: value >$, we require the following operations to be defined.
  - Insert:
    - $D[key] = value$
  - Delete:
    - $delete(D[key])$
  - Search:
    - $value = D[key], value == nil$ if key has not been stored.
- Note that the dictionary behaves like an array with non-integer index.

Ubiquity

- Dictionaries form a part of every modern computer
- language:
  - C++:
    - $Std::map < key\_type, value\_type > \ dictionary\_name;$
  - Java:
    - $Map\ dictionary\_name\ =\ new\ Hashtable();$
    - $Map\ dictionary\_name\ =\ new\ HashMap();$
    - $Map\ dictionary\_name\ =\ new\ LinkedHashMap();$
  - Python:
    - Dictionary data type – created by reference.
    - E.g. $en\_fr\ =\ \{"red" :\ "rouge", "green" :\ "vert", "blue" : "bleu", "yellow": "jaune"\}$

Motivation

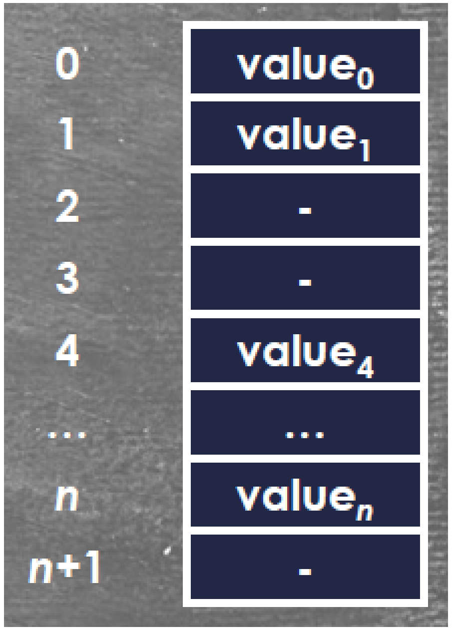
- Dictionaries are used in many applications:
  - Databases;
    - Fast access to record given key
  - Compilers;
    - Maintenance of symbol table
  - Network routers;
    - Looking up IP address
  - String matching
    - Genetic analysis.
  - Security
    - Password checking.

Implementation

- There are several ways to implement the dictionary data type:
- Let us start with the simplest (and, in most cases, worst) approach:
  - The Direct Access Table:

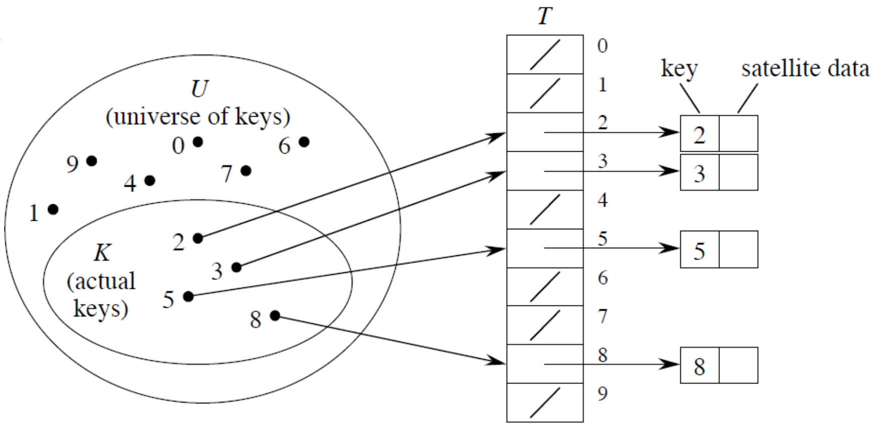
Implementation 0: The Direct Access Table

- This is simply a big array where the index of the array is the key and the contents of the array is the value.
- Only works if keys are integers
  - E.g. key = phone number, value = name.
- So, should we use it in this case?
  - Typical phone number:
  - +61 2 4221 4033
  - 11 digits – one hundred billion possible entries
  - 20 characters per name
  - Two terabytes of storage
    - For 100,000,000,000 numbers
- If U is the universe of keys $n = |U|$.



Implementation 0: The Direct Access Table

- Operations
  - $Search\ (T, K)\ \{return\ T[k]\}$
  - $Insert\ (T, x)\ \{T[key[x]] \leftarrow x\}$
  - $Delete(T, x)\ \{\ T(key[x]] \leftarrow nil\}$
  - All in $O(1)$



- Problems – when U is large
  - Keys must be non-negative integers
  - The set K of keys actually stored is small. Much space is wasted

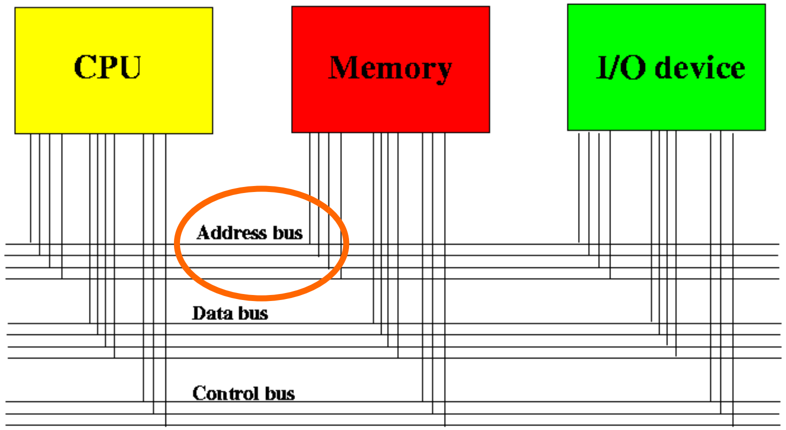
Fixing the Problems

- Problem 1: Keys must be (non-negative) integers.
- Solution: define a function $prehash(key)$: $integer$
  - This function, when given a key of whatever type we need to store returns a non-negative integer value.
  - So $D[key] = value$ becomes $T[prehash(key)] = value$.
    - T is the direct access table we are using to implement the dictionary, D.

- Hold on!
  - That was too easy!
  - What exactly does $prehash()$ do?

Implementing *prehash*()

- In theory:
  - Every piece of data in a computer is a sequence of bits
  - Every sequence of bits can be interpreted as a non-negative integer.
  - Problem solved!
  - Really?
  - Consider 8-character keys:
    - 8 characters = 64 bits
  - Does this mean we need an array with 2^{64}entries?



Implementing $prehash()$

- In Practice:
  - There are many different possible $prehash$ functions.
- Ideally:
  - $prehash(x) = prehash(y)\ iff\ x = y$
  - This is not usually always true, sometimes two different keys may have the same $prehash$ value.
  - For the sake of simplicity we will assume that the above relationship holds.

Fixing the Problems

- Problem 2:
  - Direct access tables are huge!
    - Phone numbers:- 10^{11}records
    - 8-letter words:- 2^{64} records
  - Clearly this is a BAD THING™
  - The problem here is the size of the universe of possible keys $|U|$.
- Solution: Hashing.
  - Reduce the (huge) universe of all possible keys down to a manageable size, m.
  - Our table will be of size m.
  - We have a hash function h so that $0 \leq h(key) < m$ for all valid keys.

Related References

- J. Trinh, Partitioning 2D Spaces: An Introduction to Quadtrees, 2020
- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 7.3
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 11