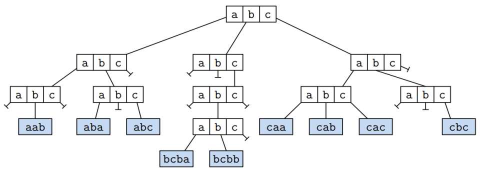
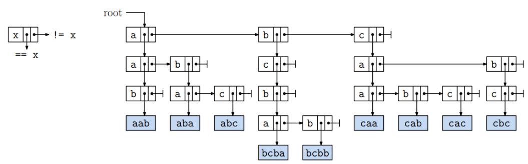
Tries/Digital Search Keys

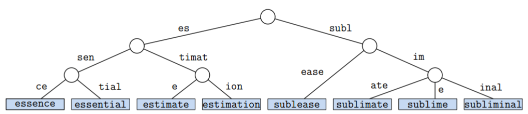
$\sum$ = Set of symbols/alphabet, each internal node has k children where $k = |\sum|$, , search time is length of word, O(1) search



de la Briandais Trees, trades off factor of k in search time for space



Path compression → Patricia Trees, O(1) query (proportion to # of characters),# nodes = # of strings, space = K*# nodes + storage for strings

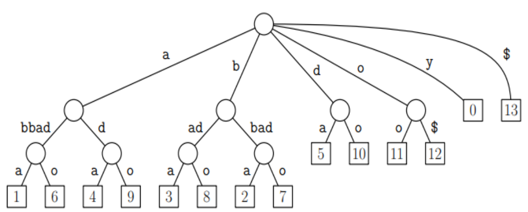


Suffix Trees

$S = "a_0 a_1 ... a_{n-1}\$"$, $ith - suffix : S_i = "a_i a_{i+1} ... a_{n-1}\$"$

For each position i, there is a minimum length substring that uniquely identifies S_i denoted as id_i. A suffix tree is a Patricia trie where we store the $n+1$ substring identifiers

times a subtring occurs = number of leaves descended from node

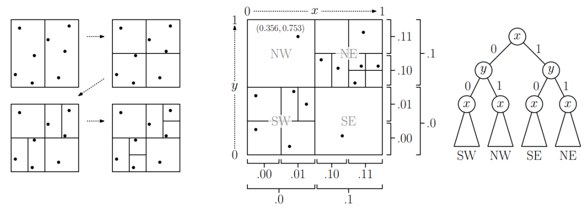


PR kd-Trees

Maps $(x, y) \mapsto$ `String`, all points lie in $[0, 1)$. $(x, y) = (0.356, 0.753)$ can be represented in binary form as $(0.01011..., 0.11000...)$

If $x = 0.a_1a_2...$ and $y = 0.b_1b_2...$ we can interleave and get $0.a_1b_1a_2b_2...$

Divide at median of cell



Euclidean MST

Spanning Tree: Acyclic subset of edges that connects all vertices together

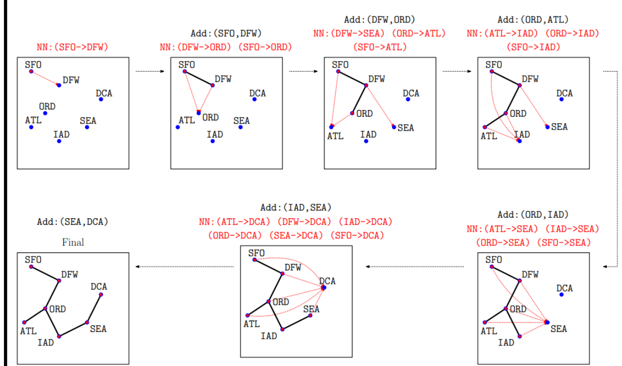
Cost of spanning tree is sum of edge weights n vertices → $n - 1$ edges

For points in R^d weights is Euclidean distance

The lowest weight edge is always safe to add

Prim's: Add point outside EMST that is closest to a point inside EMST Repeated n-1 times

Dependents List: p_k depends on $p_j \in P \backslash S$, if p_j is nn of p_k. Set of all points in S that depends on p_j is in list $dep(p_j)$



No good upperbound, could be $O(n), O(logn)$, general: $O(n * c(n)logn)$

$c(n)$ is the average number of times each point needs to update its nn

Every point inside computes it's nn outside

FinalPrac

Total space $= \sum_{i=0}^{\infty} n/2^i = 2n$ AVL: Delete max $O(log(n))$, insert max 2

Finger Search: Start at some arbitrary node x instead of the root

If x is close to search would be faster than root start.

```
void printMaxK(Node p, int k) { // print max k
    if (p != null && k > 0) // something to print?
    int rightSize = (p.right == null ? 0 : p.right.size)
    int remainder = k - rightSize // remainder after p.right
    if (remainder > 0)
        printMaxK(p.left, remainder - 1) // print left keys
        print(p.key) // print this node
    printMaxK(p.right, k) // print right keys

int printEvenOdd(Node p, int index) {
    if (p == null) return index // nothing to print
    else
        index = printEvenOdd(p.left, index) // print left subtree
        if (index % 2 == 1) print(p.key) // print current if odd
        index += 1
        return printEvenOdd(p.right, index) // print the right
}

(void*) compact(void* start, void* end) { // compact memory from start to end
    void* p = start; // p points to source block
    void* q = start; // q points to destination block
    while (p < end) {
        if (p.inUse) { // allocated block?
            memcpy(q, p, p.size); // copy to destination
            q.prevInUse = 1; // previous block is in-use
            q += p.size; // increment destination pointer
            // (no need to set q.size or q.inUse, since they are copied from p
        }
        p += p.size; // advance to the next block
    }
    // everything copied - now q points to the remaining available block
    q.inUse = 0; // this block is available
    q.prevInUse = 1; // previous block is in-use
    int blockSize = p - q; // size of this final block
    q.size = blockSize; // set q.size
    *(q + q.size -1) = blockSize; // ... and q.size2
    return q; // return pointer to this block
}
```

Kd-Tree

Deletion: Find minimum node in the right subtree with the minimum x/y according to cutdim. If right is empty, find min node in left and make left the new right. This can lead to $O(\sqrt{n})$ height over large insert/deletes

Range-Trees

1d range: o(n) space o(logn+k) count/report, k = # of ponits in range

2-D Range Tree: Space $O(nlog^{d-1}(n))$, Counting: $O(log^d(n))$

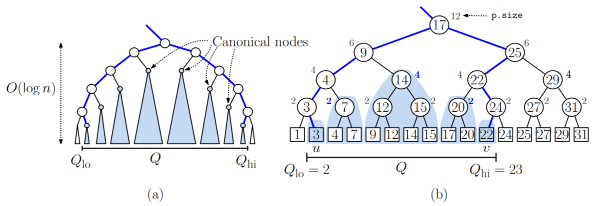
Reporting: $O(k + log^d(n))$, 1-d Range search: extended BST

Canonical Subset: S(p) = leaves of roots descended from node p

p is relevant if $S(p) \subseteq Q$, p is canonical if p is relevant but parent is not

S(p) is canonical subset and each csubset is disjoint

partial overlap: $(x_0, p.x)(p.x, x_1)$



2d range tree: each internal node p stores p.aux, a 1d y-tree for S(p)

1 point sits in multiple S(p)s', (ancestors), x-tree + (n-1) y-trees

n-1 b/c n external nodes means n-1 internal ndoes with S(p)

In d-dim, $O(nlog^d(n))$ space, $O(log^d(n))$ counting query, $O(k + log^d(n))$

Skewed Rec: $y = x + (qy^- - qx^-)$, $p' = (p_x, p_y) \mapsto (p_x, p_y - p_x)$

$p_x + (q_y^- - q_x^-) \leq p_y \leq p_x + (q_y^+ - q_x^+) \implies (q_y^- - q_x^-) \leq p_y - p_x \leq (q_y^+ - q_x^+)$

Build std range tree for points p' and ans

$q_x^- \leq x \leq q_x^+, (q_y^- - q_x^-) \leq y \leq (q_y^+ - q_x^+)$

NE Right Triangle: $z = x + y$, $p = (p_x, p_y) = (p_x, p_y, p_x + p_y)$

Build 3d range tree,l length of tri sides $\quad q_x \leq x \leq q_x + l, q_y \leq y \leq q_y + l$

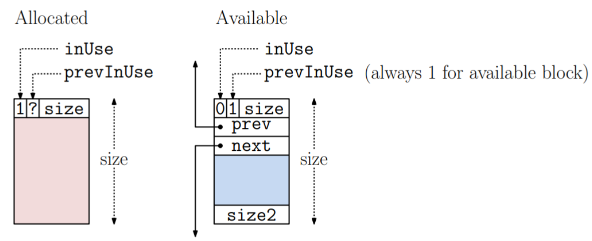
$q_x + q_y \leq z \leq q_x + q_y + l$

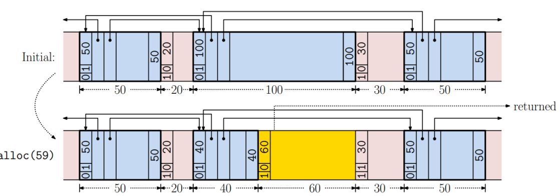
Memory Allocation Cont.

```
(void*) alloc(int b) { // allocate block with b words
    b += 1; // extra space for system overhead
    p = search available space list for block of size at least
    if (p == null) { ...Error! Insufficient memory...}
    if (p.size - b < TOO_SMALL) { // remaining fragment too sma
        avail.unlink(p); // remove entire block from avail list
        q = p; // this is block to return
    }
    else { // split the block
        p.size -= b; // decrease size by b
        *(p + p.size - 1) = p.size; // set new block's size2 fi
        q = p + p.size; // offset of start of new block
        q.size = b; // size of new block
        q.prevInUse = 0; // previous block is unused
    }
    q.inUse = 1; // new block is used
    (q + q.size).prevInUse = 1; // adjust prevInUse for followi
    return q + 1; // offset the link (to avoid header)
}
```

Memory Allocation

Block Structure: size2 access: *(p + p.size - 1), should never have 2 available consecutive blocks



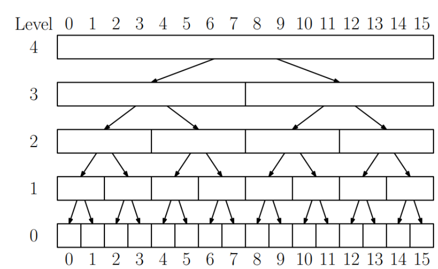


If the size of fragment is too small, we allocate entire block instead

Buddy System: internal frag is an issue, block of memory are powers of 2

If request 4 bytes, add 1 for header and round up to next highest power of 2.

Blocks of 2^k starts at addresses 2^k

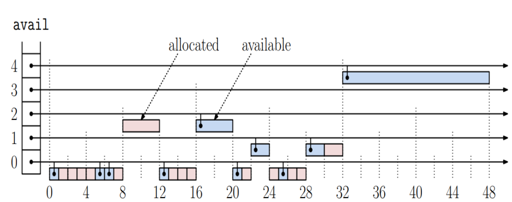


block b of size 2^k with address x, it's buddy is located at:

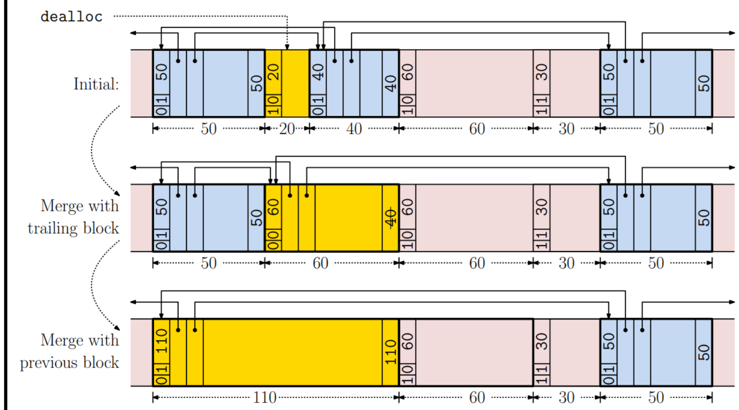
$buddy_k(x) = \begin{cases} x + 2^k & \text{if } 2^{k+1} \text{divides } x \\ x - 2^k & \text{otherwise} \end{cases} = (1 << k) \wedge x$ bitwise

Complement the k+1 bit (index starting at 1)

Buddy system can cause internal fragmentation bc of rounding the size to the next power of 2



Memory Allocation Cont.



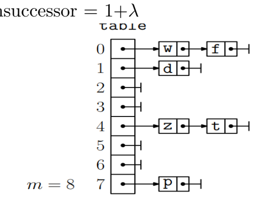
Hashing Pt.2

Scale Chaining: Store colliding entries in a separate linked list.

If table is size m, and n keys, the load factor $\lambda = n/m$, λ keys in each list

Successful search = 1+λ/2, unsuccessor = 1+λ

```
insert("d")    h("d") = 1
insert("z")    h("z") = 4
insert("p")    h("p") = 7
insert("w")    h("w") = 0
insert("t")    h("t") = 4
insert("f")    h("f") = 0
```



Open Addressing: Probe until we find an empty location

Linear Probing: Primary clustering, keys hash to same general location which forms clumps. $S_{LP} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right), U_{LP} = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

Quadratic Probe: $h(x) + i^2, i = 1, 2, ...$ $\quad i^2 = (i-1)^2 + 2i - 1$

Causes secondary clustering, clumping far away from hash location

Since probe sequence is the same. table size m is prime $\rightarrow \lfloor m/2 \rfloor$ are distinct

Double Hashing: $h(x) + g(x), h(x) + 2g(x), h(x) + 3g(x)$

m and $g(x)$ should be relatively prime. $S_{DH} = \frac{1}{\lambda} ln \frac{1}{1-\lambda}, U_{DH} = \frac{1}{1-\lambda}$

new special value "deleted", stop when reach "empty" cell

Misc

```
sibling: (1<<k)^x
parent: (~(1<<k) & x)
right: (1<<(k-1))^x
```