湖南大學

数据结构
讨论课课前资料

题　　目：	第五次讨论课（第十周）
学生姓名：	魏子铖
学生学号：	201726010308
专业班级：	软件 1703
完成时间：	2018.11.6

B-tree[edit]

Main article: B-tree

B-trees are generalizations of binary search trees in that they can have a variable number of subtrees at each node. While child-nodes have a pre-defined range, they will not necessarily be filled with data, meaning B-trees can potentially waste some space. The advantage is that B-trees do not need to be re-balanced as frequently as other self-balancing trees.

Due to the variable range of their node length, B-trees are optimized for systems that read large blocks of data. They are also commonly used in databases.

The time complexity for searching a B-tree is O(log n).

(a,b)-tree[edit]

Main article: (a,b)-tree

An (a,b)-tree is a search tree where all of its leaves are the same depth. Each node has at least **a** children and at most **b** children, while the root has at least 2 children and at most **b** children.

a and **b** can be decided with the following formula:[2]

The time complexity for searching an (a,b)-tree is O(log n).

Ternary search tree[edit]

Main article: Ternary search tree

A ternary search tree is a type of tree that can have 3 nodes: a lo kid, an equal kid, and a hi kid. Each node stores a single character and the tree itself is ordered the same way a binary search tree is, with the exception of a possible third node.

Searching a ternary search tree involves passing in a string to test whether any path contains it.

The time complexity for searching a balanced ternary search tree is O(log n).

Searching Algorithms[edit]

Searching for a Specific Key[edit]

Assuming the tree is ordered, we can take a key and attempt to locate it within the tree. The following algorithms are generalized for binary search trees, but the same idea can be applied to trees of other formats.

Recursive[edit]

```
search-recursive(key, node)
    if node is NULL
        return EMPTY_TREE
    if key < node.key
        return search-recursive(key, node.left)
    else if key > node.key
        return search-recursive(key, node.right)
    else
        return node
```

Iterative[edit]

```
searchIterative(key, node)
    currentNode := node
    while currentNode is not NULL
        if currentNode.key = key
            return currentNode
        else if currentNode.key > key
            currentNode := currentNode.left
        else
            currentNode := currentNode.right
```

Searching for Min and Max[edit]

In a sorted tree, the minimum is located at the node farthest left, while the maximum is located at the node farthest right.[3]

Minimum[edit]

```
findMinimum(node)
    if node is NULL
        return EMPTY_TREE
    min := node
    while min.left is not NULL
        min := min.left
    return min.key
```

Maximum[edit]

```
findMaximum(node)
    if node is NULL
        return EMPTY_TREE
    max := node
```

```
while max.right is not NULL
    max := max.right
return max.key
```

Operations[edit]

The common operations involving heaps are:

Basic

- *find-max* [or *find-min*]: find a maximum item of a max-heap, or a minimum item of a min-heap, respectively (a.k.a. *peek*)
- *insert*: adding a new key to the heap (a.k.a., *push*[4])
- *extract-max* [or *extract-min*]: returns the node of maximum value from a max heap [or minimum value from a min heap] after removing it from the heap (a.k.a., *pop*[5])
- *delete-max* [or *delete-min*]: removing the root node of a max heap [or min heap], respectively
- *replace*: pop root and push a new key. More efficient than pop followed by push, since only need to balance once, not twice, and appropriate for fixed-size heaps.[6]

Creation

- *create-heap*: create an empty heap
- *heapify*: create a heap out of given array of elements
- *merge* (*union*): joining two heaps to form a valid new heap containing all the elements of both, preserving the original heaps.
- *meld*: joining two heaps to form a valid new heap containing all the elements of both, destroying the original heaps.

Inspection

- *size*: return the number of items in the heap.
- *is-empty*: return true if the heap is empty, false otherwise.

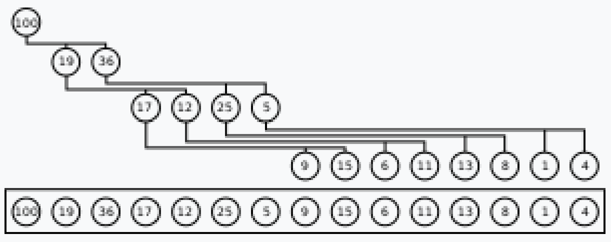
Internal

- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *delete*: delete an arbitrary node (followed by moving last node and sifting to maintain heap)
- *sift-up*: move a node up in the tree, as long as needed; used to restore heap condition after insertion. Called "sift" because node moves up the tree until it reaches the correct level, as in a sieve.

- *sift-down*: move a node down in the tree, similar to sift-up; used to restore heap condition after deletion or replacement.

Implementation[edit]

Heaps are usually implemented in an array (fixed size or dynamic array), and do not require pointers between elements. After an element is inserted into or deleted from a heap, the heap property may be violated and the heap must be balanced by internal operations.



Example of a complete binary max-heap with node keys being integers from 1 to 100 and how it would be stored in an array.

Binary heaps may be represented in a very space-efficient way (as an implicit data structure) using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at position *n* would be at positions **2n** and **2n + 1** in a one-based array, or **2n + 1**and **2n + 2** in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by sift-up or sift-down operations (swapping elements which are out of order). As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

Different types of heaps implement the operations in different ways, but notably, insertion is often done by adding the new element at the end of the heap in the first available free space. This will generally violate the heap property, and so the elements are then sifted up until the heap property has been reestablished. Similarly, deleting the root is done by removing the root and then putting the last element in the root and sifting down to rebalance. Thus replacing is done by deleting the root and putting the *new* element in the root and sifting down, avoiding a sifting up step compared to pop (sift down of last element) followed by push (sift up of new element).

Construction of a binary (or *d*-ary) heap out of a given array of elements may be performed in linear time using the classic Floyd algorithm, with the worst-case number of comparisons equal to $2N - 2s_2(N) - e_2(N)$ (for a binary heap), where $s_2(N)$ is the sum of all digits of the binary representation of N and $e_2(N)$ is the exponent of 2 in the prime factorization of N.[7] This is faster than a sequence of consecutive insertions into an originally empty heap, which is log-linear.[a]

Variants[edit]

- 2–3 heap
- B-heap
- Beap
- Binary heap
- Binomial heap
- Brodal queue
- *d*-ary heap
- Fibonacci heap
- Leaf heap
- Leftist heap
- Pairing heap
- Radix heap
- Randomized meldable heap
- Skew heap
- Soft heap
- Ternary heap
- Treap
- Weak heap

Comparison of theoretic bounds for variants[edit]

In the following time complexities[8] $O(f)$ is an asymptotic upper bound and $\Theta(f)$ is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	Binary[8]	Leftist	Binomial[8]	Fibonacci[8][9]	Pairing[10]	Brodal[11][b]	Rank-pairing[13]	Strict Fibonacci[14]	2-3 heap
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	?
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$	$O(\log n)$[c]
insert	$O(\log$	$\Theta(\text{lo}$	$\Theta(1)$[c]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(\log$

	n)	g n)							n)[c]
decrease-key	Θ(log n)	$\Theta(n)$	Θ(log n)	$\Theta(1)$[c]	o(log n)[c][d]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$	$\Theta(1)$
merge	$\Theta(n)$	Θ(log n)	O(log n)[e]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	?

1. **Jump up^** Each insertion takes O(log(k)) in the existing size of the heap, thus . Since , a constant factor (half) of these insertions are within a constant factor of the maximum, so asymptotically we can assume ; formally the time is . This can also be readily seen from Stirling's approximation.
2. **Jump up^** Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with n elements can be constructed bottom-up in $O(n)$.[12]
3. ^ Jump up to:*a* *b* *c* *d* *e* *f* *g* *h* *i* Amortized time.
4. **Jump up^** Lower bound of [15] upper bound of [16]
5. **Jump up^** n is the size of the larger heap.

Applications[edit]

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Selection algorithms: A heap allows access to the min or max element in constant time, and other selections (such as median or kth-element) can be done in sub-linear time on data that is in a heap.[17]
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal-spanning-tree algorithm and Dijkstra's shortest-path algorithm.
- Priority Queue: A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods.
- K-way merge: A heap data structure is useful to merge many already-sorted input streams into a single sorted output stream. Examples of the need for merging include

external sorting and streaming results from distributed data such as a log structured merge tree. The inner loop is obtaining the min element, replacing with the next element for the corresponding input stream, then doing a sift-down heap operation. (Alternatively the replace function.) (Using extract-max and insert functions of a priority queue are much less efficient.)

- Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.

Implementations[edit]

- The C++ Standard Library provides the `make_heap`, `push_heap` and `pop_heap` algorithms for heaps (usually implemented as binary heaps), which operate on arbitrary random access iterators. It treats the iterators as a reference to an array, and uses the array-to-heap conversion. It also provides the container adaptor `priority_queue`, which wraps these facilities in a container-like class. However, there is no standard support for the replace, sift-up/sift-down, or decrease/increase-key operations.
- The Boost C++ libraries include a heaps library. Unlike the STL, it supports decrease and increase operations, and supports additional types of heap: specifically, it supports *d*-ary, binomial, Fibonacci, pairing and skew heaps.
- There is a generic heap implementation for C and C++ with D-ary heap and B-heap support. It provides an STL-like API.
- The standard library of the D programming language includes `std.container.BinaryHeap`, which is implemented in terms of D's ranges. Instances can be constructed from any random-access range. `BinaryHeap` exposes an input range interface that allows iteration with D's built-in `foreach` statements and integration with the range-based API of the `std.algorithm` package.
- The Java platform (since version 1.5) provides a binary heap implementation with the class `java.util.PriorityQueue` in the Java Collections Framework. This class implements by default a min-heap; to implement a max-heap, programmer should write a custom comparator. There is no support for the replace, sift-up/sift-down, or decrease/increase-key operations.
- Python has a `heapq` module that implements a priority queue using a binary heap. The library exposes a heapreplace function to support k-way merging.
- PHP has both max-heap (`SplMaxHeap`) and min-heap (`SplMinHeap`) as of version 5.3 in the Standard PHP Library.
- Perl has implementations of binary, binomial, and Fibonacci heaps in the `Heap` distribution available on CPAN.
- The Go language contains a `heap` package with heap algorithms that operate on an arbitrary type that satisfies a given interface. That package does not support the replace, sift-up/sift-down, or decrease/increase-key operations.

- Apple's Core Foundation library contains a `CFBinaryHeap` structure.
- Pharo has an implementation of a heap in the Collections-Sequenceable package along with a set of test cases. A heap is used in the implementation of the timer event loop.
- The Rust programming language has a binary max-heap implementation, `BinaryHeap`, in the `collections` module of its standard library.

History and etymology[edit]

Tries were first described by René de la Briandais in 1959.[1][2]:336 The term *trie* was coined two years later by Edward Fredkin, who pronounces it /ˈtriː/ (as "tree"), after the middle syllable of *retrieval*.[3][4] However, other authors pronounce it /ˈtraɪ/ (as "try"), in an attempt to distinguish it verbally from "tree".[3][4][5]

Applications[edit]

As a replacement for other data structures[edit]

As discussed below, a trie has a number of advantages over binary search trees.[6] A trie can also be used to replace a hash table, over which it has the following advantages:

- Looking up data in a trie is faster in the worst case, O(m) time (where m is the length of a search string), compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash.[*citation needed*]
- There are no collisions of different keys in a trie.
- Buckets in a trie, which are analogous to hash table buckets that store key collisions, are necessary only if a single key is associated with more than one value.
- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Trie lookup can be slower in some cases than hash tables, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory.[7]

- Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless, a bitwise trie can handle standard IEEE single and double format floating point numbers.[citation needed]
- Some tries can require more space than a hash table, as memory may be allocated for each character in the search string, rather than a single chunk of memory for the whole entry, as in most hash tables.

Dictionary representation[edit]

A common application of a trie is storing a predictive text or autocomplete dictionary, such as found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e., storage of information auxiliary to each word is not required), a minimal deterministic acyclic finite state automaton (DAFSA) would use less space than a trie. This is because a DAFSA can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored.

Tries are also well suited for implementing approximate matching algorithms,[8] including those used in spell checking and hyphenation[4] software.

Term indexing[edit]

A discrimination tree term index stores its information in a trie data structure.[9]

Algorithms[edit]

The trie is a tree of nodes which supports Find and Insert operations. Find returns the value for a key string, and Insert inserts a string (the key) and a value into the trie. Both Insert and Find run in $O(n)$ time, where n is the length of the key.

A simple Node class can be used to represent nodes in the trie:

```
class Node():
    def __init__(self):
        # Note that using dictionary for children (as in
this implementation) would not allow lexicographic
sorting mentioned in the next section (Sorting),
        # because ordinary dictionary would not preserve
the order of the keys
        self.children = {}  # mapping from character ==>
Node
        self.value = None
```

Note that `children` is a dictionary of characters to a node's children; and it is said that a "terminal" node is one which represents a complete string.
A trie's value can be looked up as follows:

```python
def find(node, key):
    for char in key:
        if char in node.children:
            node = node.children[char]
        else:
            return None
    return node.value
```

Insertion proceeds by walking the trie according to the string to be inserted, then appending new nodes for the suffix of the string that is not contained in the trie:

```python
def insert(root, string, value):
    node = root
    index_last_char = None
    for index_char, char in enumerate(string):
        if char in node.children:
            node = node.children[char]
        else:
            index_last_char = index_char
            break

    # append new nodes for the remaining characters, if
any
    if index_last_char is not None:
        for char in string[index_last_char:]:
            node.children[char] = Node()
            node = node.children[char]

    # store value in the terminal node
    node.value = value
```

Sorting[edit]

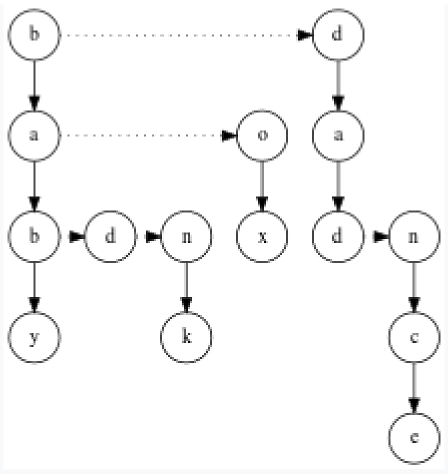
Lexicographic sorting of a set of keys can be accomplished by building a trie from them, and traversing it in pre-order, printing only the leaves' values. This algorithm is a form of radix sort.[citation needed]

A trie forms the fundamental data structure of Burstsort, which (in 2007) was the fastest known string sorting algorithm.[10] However, now there are faster string sorting algorithms.[11]

Full text search[edit]

A special kind of trie, called a suffix tree, can be used to index all suffixes in a text in order to carry out fast full text searches.

Implementation strategies[edit]



A trie implemented as a doubly chained tree: vertical arrows are `child` pointers, dashed horizontal arrows are `next` pointers. The set of strings stored in this trie is `{baby, bad, bank, box, dad, dance}`. The lists are sorted to allow traversal in lexicographic order.

There are several ways to represent tries, corresponding to different trade-offs between memory use and speed of the operations. The basic form is that of a linked set of nodes, where each node contains an array of child pointers, one for each symbol in the alphabet (so for the English alphabet, one would store 26 child pointers and for the alphabet of bytes, 256 pointers). This is simple but wasteful in terms of memory: using the alphabet of bytes (size 256) and four-byte pointers, each node requires a kilobyte of storage, and when there is little overlap in the strings' prefixes, the number of required nodes is roughly the combined length of the stored strings.[2]:341 Put another way, the nodes near the bottom of the tree tend to have few children and there are many of them, so the structure wastes space storing null pointers.[12]

The storage problem can be alleviated by an implementation technique called *alphabet reduction*, whereby the original strings are reinterpreted as longer strings over a smaller alphabet. E.g., a string of *n* bytes can alternatively be regarded as a string of $2n$ four-bit units and stored in a trie with sixteen pointers per node. Lookups need to visit twice as many nodes in the worst case, but the storage requirements go down by a factor of eight.[2]:347–352

An alternative implementation represents a node as a triple `(symbol, child, next)` and links the children of a node together as a singly linked list: `child` points to the node's first child, `next` to the parent node's next child.[12][13] The set of children can also be represented as a binary search tree; one instance of this idea is the ternary search tree developed by Bentley and Sedgewick.[2]:353

Another alternative in order to avoid the use of an array of 256 pointers (ASCII), as suggested before, is to store the alphabet array as a bitmap of 256 bits representing the ASCII alphabet, reducing dramatically the size of the nodes.[14]

Bitwise tries[edit]

This section **does not cite any sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(February 2015)* *(Learn how and when to remove this template message)*

Bitwise tries are much the same as a normal character-based trie except that individual bits are used to traverse what effectively becomes a form of binary tree. Generally, implementations use a special CPU instruction to very quickly find the first set bit in a fixed length key (e.g., GCC's `__builtin_clz()` intrinsic). This value is then used to index a 32- or 64-entry table which points to the first item in the bitwise trie with that number of leading zero bits. The search then proceeds by testing each subsequent bit in the key and choosing `child[0]` or `child[1]` appropriately until the item is found.

Although this process might sound slow, it is very cache-local and highly parallelizable due to the lack of register dependencies and therefore in fact has excellent performance on modern out-of-order execution CPUs. A red-black tree for example performs much better on paper, but is highly cache-unfriendly and causes multiple pipeline and TLB stalls on modern CPUs which makes that algorithm bound by memory latency rather than CPU speed. In comparison, a bitwise trie rarely accesses memory, and when it does, it does so only to read, thus avoiding SMP cache coherency overhead. Hence, it is increasingly becoming the algorithm of choice for code that performs many rapid insertions and deletions, such as memory allocators (e.g., recent versions of the famous Doug Lea's allocator (dlmalloc) and its descendents).

Compressing tries[edit]

Compressing the trie and merging the common branches can sometimes yield large performance gains. This works best under the following conditions:

- The trie is mostly static (key insertions to or deletions from a pre-filled trie are disabled).[citation needed]

- Only lookups are needed.
- The trie nodes are not keyed by node-specific data, or the nodes' data are common.[15]
- The total set of stored keys is very sparse within their representation space.[citation needed]

For example, it may be used to represent sparse bitsets, i.e., subsets of a much larger, fixed enumerable set. In such a case, the trie is keyed by the bit element position within the full set. The key is created from the string of bits needed to encode the integral position of each element. Such tries have a very degenerate form with many missing branches. After detecting the repetition of common patterns or filling the unused gaps, the unique leaf nodes (bit strings) can be stored and compressed easily, reducing the overall size of the trie.

Such compression is also used in the implementation of the various fast lookup tables for retrieving Unicode character properties. These could include case-mapping tables (e.g. for the Greek letter pi, from Π to π), or lookup tables normalizing the combination of base and combining characters (like the a-umlaut in German, ä, or the dalet-patah-dagesh-ole in Biblical Hebrew, דַּ֫). For such applications, the representation is similar to transforming a very large, unidimensional, sparse table (e.g. Unicode code points) into a multidimensional matrix of their combinations, and then using the coordinates in the hyper-matrix as the string key of an uncompressed trie to represent the resulting character. The compression will then consist of detecting and merging the common columns within the hyper-matrix to compress the last dimension in the key. For example, to avoid storing the full, multibyte Unicode code point of each element forming a matrix column, the groupings of similar code points can be exploited. Each dimension of the hyper-matrix stores the start position of the next dimension, so that only the offset (typically a single byte) need be stored. The resulting vector is itself compressible when it is also sparse, so each dimension (associated to a layer level in the trie) can be compressed separately.

Some implementations do support such data compression within dynamic sparse tries and allow insertions and deletions in compressed tries. However, this usually has a significant cost when compressed segments need to be split or merged. Some tradeoff has to be made between data compression and update speed. A typical strategy is to limit the range of global lookups for comparing the common branches in the sparse trie.[citation needed]

The result of such compression may look similar to trying to transform the trie into a directed acyclic graph (DAG), because the reverse transform from a DAG to a trie is obvious and always possible. However, the shape of the DAG is determined by the form of the key chosen to index the nodes, in turn constraining the compression possible.

Another compression strategy is to "unravel" the data structure into a single byte array.[16] This approach eliminates the need for node pointers, substantially reducing the memory requirements. This in turn permits memory mapping and the use of virtual memory to efficiently load the data from disk.

One more approach is to "pack" the trie.[4] Liang describes a space-efficient implementation of a sparse packed trie applied to automatic hyphenation, in which the descendants of each node may be interleaved in memory.

External memory tries[edit]

Several trie variants are suitable for maintaining sets of strings in external memory, including suffix trees. A combination of trie and B-tree, called the *B-trie* has also been suggested for this task; compared to suffix trees, they are limited in the supported operations but also more compact, while performing update operations faster.[17]

Overview[edit]

Binary space partitioning is a generic process of recursively dividing a scene into two until the partitioning satisfies one or more requirements. It can be seen as a generalisation of other spatial tree structures such as *k*-d trees and quadtrees, one where hyperplanes that partition the space may have any orientation, rather than being aligned with the coordinate axes as they are in *k*-d trees or quadtrees. When used in computer graphics to render scenes composed of planar polygons, the partitioning planes are frequently chosen to coincide with the planes defined by polygons in the scene.

The specific choice of partitioning plane and criterion for terminating the partitioning process varies depending on the purpose of the BSP tree. For example, in computer graphics rendering, the scene is divided until each node of the BSP tree contains only polygons that can render in arbitrary order. When back-face culling is used, each node therefore contains a convex set of polygons, whereas when rendering double-sided polygons, each node of the BSP tree contains only polygons in a single plane. In collision detection or ray tracing, a scene may be divided up into primitives on which collision or ray intersection tests are straightforward.

Binary space partitioning arose from the computer graphics need to rapidly draw three-dimensional scenes composed of polygons. A simple way to draw such scenes is the painter's algorithm, which produces polygons in order of distance from the viewer, back to front, painting over the background and previous polygons with each closer object. This approach has two disadvantages: time required to sort polygons in back to front order, and the possibility of errors in overlapping polygons. Fuchs and co-authors[2] showed that constructing a BSP tree solved both of these problems by providing a rapid method of sorting polygons with respect to a given viewpoint (linear in the number of polygons in the scene) and by subdividing overlapping polygons to avoid errors that can occur with the painter's algorithm. A disadvantage of binary space partitioning is that generating a BSP tree can be time-consuming. Typically, it is therefore performed once on static geometry, as a pre-calculation step, prior to rendering or other realtime operations on a scene. The expense of constructing a BSP tree makes it difficult and inefficient to directly implement moving objects into a tree.

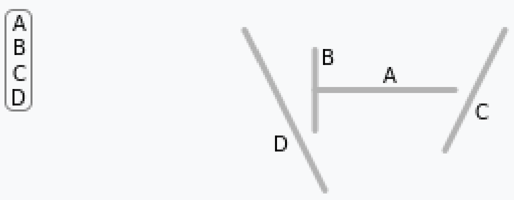
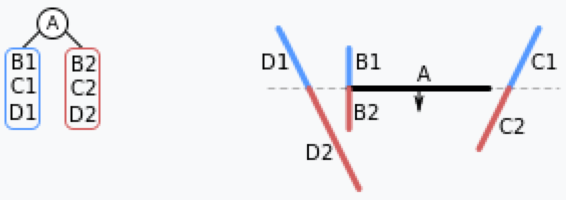
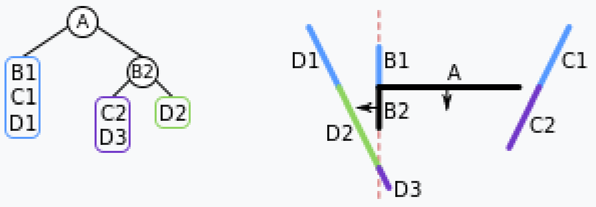
BSP trees are often used by 3D video games, particularly first-person shooters and those with indoor environments. Game engines using BSP trees include the Doom, Quake, and Source engines. In them, BSP trees containing the static geometry of a scene are often used together with a Z-buffer, to correctly merge movable objects such as doors and characters onto the background scene. While binary space partitioning provides a convenient way to store and retrieve spatial information about polygons in a scene, it does not solve the problem of visible surface determination.

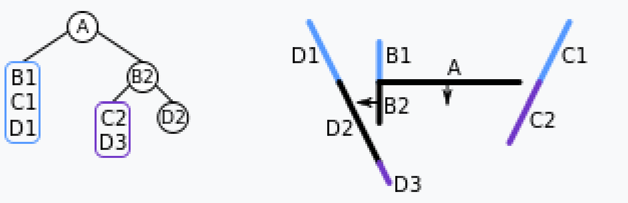
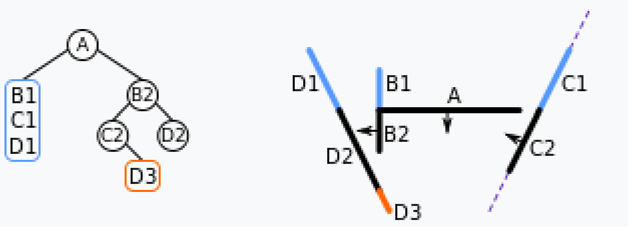
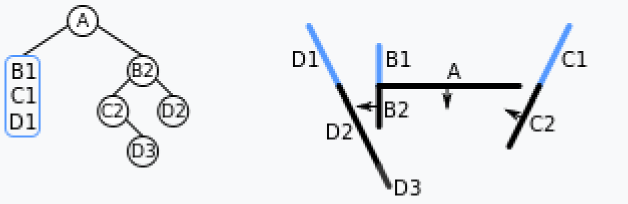
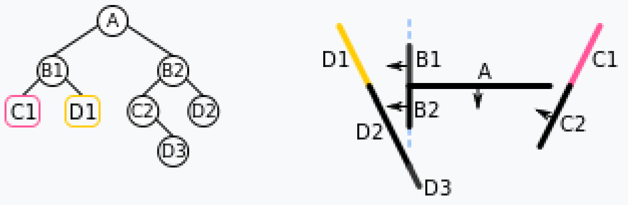
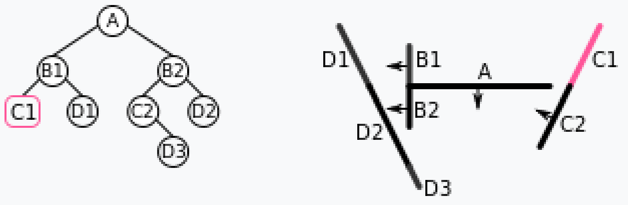
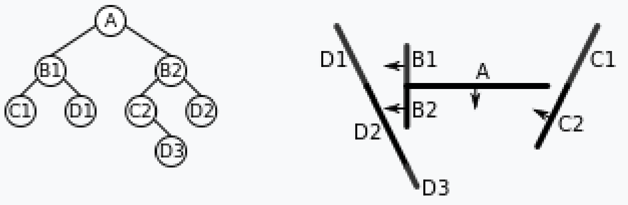
Generation[edit]

The canonical use of a BSP tree is for rendering polygons (that are double-sided, that is, without back-face culling) with the painter's algorithm. Each polygon is designated with a front side and a back side which could be chosen arbitrarily and only affects the structure of the tree but not the required result.[2] Such a tree is constructed from an unsorted list of all the polygons in a scene. The recursive algorithm for construction of a BSP tree from that list of polygons is:[2]

1. Choose a polygon *P* from the list.
2. Make a node *N* in the BSP tree, and add *P* to the list of polygons at that node.
3. For each other polygon in the list:
    1. If that polygon is wholly in front of the plane containing *P*, move that polygon to the list of nodes in front of *P*.
    2. If that polygon is wholly behind the plane containing *P*, move that polygon to the list of nodes behind *P*.
    3. If that polygon is intersected by the plane containing *P*, split it into two polygons and move them to the respective lists of polygons behind and in front of *P*.
    4. If that polygon lies in the plane containing *P*, add it to the list of polygons at node *N*.
4. Apply this algorithm to the list of polygons in front of *P*.
5. Apply this algorithm to the list of polygons behind *P*.

The following diagram illustrates the use of this algorithm in converting a list of lines or polygons into a BSP tree. At each of the eight steps (i.-viii.), the algorithm above is applied to a list of lines, and one new node is added to the tree.

	Start with a list of lines, (or in 3D, polygons) making up the scene. In the tree diagrams, lists are denoted by rounded rectangles and nodes in the BSP tree by circles. In the spatial diagram of the lines, the direction chosen to be the 'front' of a line is denoted by an arrow.	
i.	Following the steps of the algorithm above, 1. We choose a line, A, from the list and,... 2. ...add it to a node. 3. We split the remaining lines in the list into those in front of A (i.e. B2, C2, D2), and those behind (B1, C1, D1). 4. We first process the lines in front of A (in steps ii–v),... 5. ...followed by those behind (in steps vi–vii).	 A B1 C1 D1 B2 C2 D2 D1 B1 A C1 B2 C2 D2
ii.	We now apply the algorithm to the list of lines in front of A (containing B2, C2, D2). We choose a line, B2, add it to a node and split the rest of the list into those lines that are in front of B2 (D2), and those that are behind it (C2, D3).	 A B1 C1 D1 B2 C2 D3 D2 D1 B1 A C1 D2 B2 C2 D3

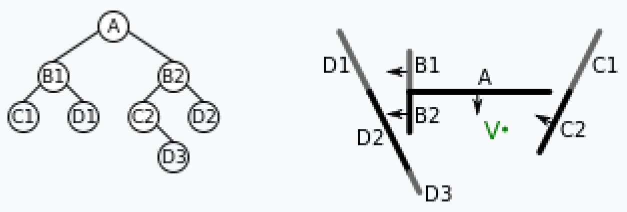
iii.	Choose a line, D2, from the list of lines in front of B2 and A. It is the only line in the list, so after adding it to a node, nothing further needs to be done.	
iv.	We are done with the lines in front of B2, so consider the lines behind B2 (C2 and D3). Choose one of these (C2), add it to a node, and put the other line in the list (D3) into the list of lines in front of C2.	
v.	Now look at the list of lines in front of C2. There is only one line (D3), so add this to a node and continue.	
vi.	We have now added all of the lines in front of A to the BSP tree, so we now start on the list of lines behind A. Choosing a line (B1) from this list, we add B1 to a node and split the remainder of the list into lines in front of B1 (i.e. D1), and lines behind B1 (i.e. C1).	
vii.	Processing first the list of lines in front of B1, D1 is the only line in this list, so add this to a node and continue.	
viii.	Looking next at the list of lines behind B1, the only line in this list is C1, so add this to a node, and the BSP tree is complete.	

The final number of polygons or lines in a tree is often larger (sometimes much larger[2]) than the original list, since lines or polygons that cross the partitioning plane must be split into two. It is desirable to minimize this increase, but also to maintain reasonable balance in the final tree. The choice of which polygon or line is used as a partitioning plane (in step 1 of the algorithm) is therefore important in creating an efficient BSP tree.

Traversal[edit]

A BSP tree is traversed in a linear time, in an order determined by the particular function of the tree. Again using the example of rendering double-sided polygons using the painter's algorithm, to draw a polygon *P* correctly requires that all polygons behind the plane *P* lies in must be drawn first, then polygon *P*, then finally the polygons in front of *P*. If this drawing order is satisfied for all polygons in a scene, then the entire scene renders in the correct order. This procedure can be implemented by recursively traversing a BSP tree using the following algorithm.[2] From a given viewing location *V*, to render a BSP tree,

1. If the current node is a leaf node, render the polygons at the current node.
2. Otherwise, if the viewing location *V* is in front of the current node:
    1. Render the child BSP tree containing polygons behind the current node
    2. Render the polygons at the current node
    3. Render the child BSP tree containing polygons in front of the current node
3. Otherwise, if the viewing location *V* is behind the current node:
    1. Render the child BSP tree containing polygons in front of the current node
    2. Render the polygons at the current node
    3. Render the child BSP tree containing polygons behind the current node
4. Otherwise, the viewing location *V* must be exactly on the plane associated with the current node. Then:
    1. Render the child BSP tree containing polygons in front of the current node
    2. Render the child BSP tree containing polygons behind the current node



Applying this algorithm recursively to the BSP tree generated above results in the following steps:

- The algorithm is first applied to the root node of the tree, node *A*. *V* is in front of node *A*, so we apply the algorithm first to the child BSP tree containing polygons behind *A*
    - This tree has root node *B1*. *V* is behind *B1* so first we apply the algorithm to the child BSP tree containing polygons in front of *B1*:

        - This tree is just the leaf node *D1*, so the polygon *D1* is rendered.
    - We then render the polygon *B1*.
    - We then apply the algorithm to the child BSP tree containing polygons behind *B1*:
        - This tree is just the leaf node *C1*, so the polygon *C1* is rendered.
- We then draw the polygons of *A*
- We then apply the algorithm to the child BSP tree containing polygons in front of *A*
    - This tree has root node *B2*. *V* is behind *B2* so first we apply the algorithm to the child BSP tree containing polygons in front of *B2*:
        - This tree is just the leaf node *D2*, so the polygon *D2* is rendered.
    - We then render the polygon *B2*.
    - We then apply the algorithm to the child BSP tree containing polygons behind *B2*:
        - This tree has root node *C2*. *V* is in front of *C2* so first we would apply the algorithm to the child BSP tree containing polygons behind *C2*. There is no such tree, however, so we continue.
        - We render the polygon *C2*.
        - We apply the algorithm to the child BSP tree containing polygons in front of *C2*
            - This tree is just the leaf node *D3*, so the polygon *D3* is rendered.

The tree is traversed in linear time and renders the polygons in a far-to-near ordering (*D1*, *B1*, *C1*, *A*, *D2*, *B2*, *C2*, *D3*) suitable for the painter's algorithm.

Timeline[edit]

- 1969 Schumacker et al.[1] published a report that described how carefully positioned planes in a virtual environment could be used to accelerate polygon ordering. The technique made use of depth coherence, which states that a polygon on the far side of the plane cannot, in any way, obstruct a closer polygon. This was used in flight simulators made by GE as well as Evans and Sutherland. However, creation of the polygonal data organization was performed manually by scene designer.

- 1980 Fuchs et al.[2] extended Schumacker's idea to the representation of 3D objects in a virtual environment by using planes that lie coincident with polygons to recursively partition the 3D space. This provided a fully automated and algorithmic generation of a hierarchical polygonal data structure known as a Binary Space Partitioning Tree (BSP Tree). The process took place as an off-line preprocessing step that was performed once per environment/object. At run-time, the view-dependent visibility ordering was generated by traversing the tree.
- 1981 Naylor's Ph.D thesis provided a full development of both BSP trees and a graph-theoretic approach using strongly connected components for pre-computing visibility, as well as the connection between the two methods. BSP trees as a

dimension independent spatial search structure was emphasized, with applications to visible surface determination. The thesis also included the first empirical data demonstrating that the size of the tree and the number of new polygons was reasonable (using a model of the Space Shuttle).

- 1983 Fuchs et al. described a micro-code implementation of the BSP tree algorithm on an Ikonas frame buffer system. This was the first demonstration of real-time visible surface determination using BSP trees.
- 1987 Thibault and Naylor[3] described how arbitrary polyhedra may be represented using a BSP tree as opposed to the traditional b-rep (boundary representation). This provided a solid representation vs. a surface based-representation. Set operations on polyhedra were described using a tool, enabling constructive solid geometry (CSG) in real-time. This was the fore runner of BSP level design using "brushes", introduced in the Quake editor and picked up in the Unreal Editor.
- 1990 Naylor, Amanatides, and Thibault provided an algorithm for merging two BSP trees to form a new BSP tree from the two original trees. This provides many benefits including: combining moving objects represented by BSP trees with a static environment (also represented by a BSP tree), very efficient CSG operations on polyhedra, exact collisions detection in O(log n * log n), and proper ordering of transparent surfaces contained in two interpenetrating objects (has been used for an x-ray vision effect).
- 1990 Teller and Séquin proposed the offline generation of potentially visible sets to accelerate visible surface determination in orthogonal 2D environments.
- 1991 Gordon and Chen [CHEN91] described an efficient method of performing front-to-back rendering from a BSP tree, rather than the traditional back-to-front approach. They utilised a special data structure to record, efficiently, parts of the screen that have been drawn, and those yet to be rendered. This algorithm, together with the description of BSP Trees in the standard computer graphics textbook of the day (*Computer Graphics: Principles and Practice*) was used by John Carmackin the making of *Doom* (video game).
- 1992 Teller's PhD thesis described the efficient generation of potentially visible sets as a pre-processing step to accelerate real-time visible surface determination in arbitrary 3D polygonal environments. This was used in *Quake* and contributed significantly to that game's performance.
- 1993 Naylor answered the question of what characterizes a good BSP tree. He used expected case models (rather than worst case analysis) to mathematically measure the expected cost of searching a tree and used this measure to build good BSP trees. Intuitively, the tree represents an object in a multi-resolution fashion (more exactly, as a tree of approximations). Parallels with Huffman codes and probabilistic binary search trees are drawn.
- 1993 Hayder Radha's PhD thesis described (natural) image representation methods using BSP trees. This includes the development of an optimal BSP-tree construction framework for any arbitrary input image. This framework is based on a new image transform, known as the Least-Square-Error (LSE) Partitioning Line (LPE) transform.

H. Radha's thesis also developed an optimal rate-distortion (RD) image compression framework and image manipulation approaches using BSP trees.

参考文献：

维基百科 --https://en.wikipedia.org/wiki/Tree_(data_structure)