sdsl Cheat Sheet

Data structures

The library code is in the `sdsl` namespace. Either import the namespace in your program (`using namespace sdsl;`) or qualify all identifieres by a `sdsl::`-prefix.
Each section corresponds to a header file. The file is hyperlinked as part of the section heading.
We have two types of data structures in sdsl. *Self-contained* and *support* structures. A support object `s` can extend a self-contained object `o` (e.g. add functionality), but requires access to `o`. Support structures contain the substring `support` in their class names.

Integer Vectors (IV)

The core of the library is the class `int_vector<`w`>`.
Parameter w corresponds to the fixed length of each element in bits. For $w = 8, 16, 32, 64, 1$ the length is fixed during compile time and the vectors correspond to `std::vector<uint`w`_t>` resp. `std::vector<bool>`. If $w = 0$ (default) the length can be set during runtime. *Constructor:* `int_vector<>(`n`,`x`,`ℓ`)`, with n equals size, x default integer value, ℓ width of integer (has no effect for $w > 0$).
Public methods: `operator[i]`, `size()`, `width()`, `data()`.

Manipulating `int_vector<`w`> v`

Method	*Description*
`v[`i`]=`x	Set entry `v[`i`]` to x.
`v.width(`ℓ`)`	Set width to ℓ, if $w = 0$.
`v.resize(`n`)`	Resize `v` to n elements.

Useful methods in namespace `sdsl::util`:

`set_to_value(v,`k`)`	Set `v[`i`]=`k for each i.
`set_to_id(v)`	Set `v[`i`]=`i for each i.
`set_random_bits(v)`	Set elements to random bits.
`mod(v,`m`)`	Set `v[`i`]=v[`i`]` mod m for each i.
`bit_compress(v)`	Gets $x = \max_i$`v[`i`]` and $\ell = \lceil \log(x-1) \rceil + 1$ and packs the entries in ℓ-bit integers.
`expand_width(v,`ℓ`)`	Expands the width of each integer to ℓ bits, if $\ell \geq$ `v.width()`.

Compressed Integer Vectors (CIV)

For a vector `v`, `enc_vector` stores the self-delimiting coded deltas (`v[`i+1`]`$-$`v[`i`]`). Fast random access is achieved by sampling values of `v` at rate `t_dens`. Available coder are `coder::elias_delta`, `coder::elias_gamma`, and `coder::fibonacci`.
Class `vlc_vector` stores each `v[i]` as self-delimiting codeword. Samples at rate `t_dens` are inserted for fast random access.

Bitvectors (BV)

Representations for a bitvector of length n with m set bits.

Class	*Description*	*Space*
`bit_vector`	plain bitvector	$64\lceil n/64+1 \rceil$
`bit_vector_il`	interleaved bitvector	$\approx n(1 + K/64)$
`rrr_vector`	H_0-compressed bitvector	$\approx \lceil \log \binom{m}{n} \rceil$
`sd_vector`	sparse bitvector	$\approx m \cdot (2 + \log \frac{n}{m})$

`bit_vector` equals `int_vector<1>` and is therefore dynamic.
Public Methods: `operator[`i`]`, `size()`, `begin()`, `end()`
Public Types: `rank_1_type`, `select_1_type`, `select_0_type`[1].
Each bitvector can be constructed out of a `bit_vector` object.

Rank Supports (RS)

RSs add rank functionality to BV. Methods `rank(`i`)` and `operator(`i`)` return the number of set bits[2] in the prefix $[0..i)$ of the supported BV for $i \in [0, n]$.

Class	*Compatible BV*	*+Bits*	*Time*
`rank_support_v`	`bit_vector`	$0.25n$	$\mathcal{O}(1)$
`rank_support_v5`	`bit_vector`	$0.0625n$	$\mathcal{O}(1)$
`rank_support_scan`	`bit_vector`	64	$\mathcal{O}(n)$
`rank_support_il`	`bit_vector_il`	128	$\mathcal{O}(1)$
`rank_support_rrr`	`rrr_vector`	80	$\mathcal{O}(k)$
`rank_support_sd`	`sd_vector`	64	$\mathcal{O}(1)$

Call `util::init_support(rs,bv)` to initialize rank structure `rs` to bitvector `bv`. Call `rs(`i`)` to get `rank(`i`)` $= \sum_{k=0}^{k<i}$ `bv[`k`]`

Select Supports (SLS)

SLSs add select functionality to BV. Let m be the number of set bits in BV. Methods `select(`i`)` and `operator(`i`)` return the position of the i-th set bit[3] in BV for $i \in [1..m]$.

Class	*Compatible BV*	*+Bits*	*Time*
`select_support_mcl`	`bit_vector`	$\leq 0.2n$	$\mathcal{O}(1)$
`select_support_scan`	`bit_vector`	64	$\mathcal{O}(n)$
`select_support_il`	`bit_vector_il`	64	$\mathcal{O}(\log n)$
`select_support_rrr`	`rrr_vector`	64	$\mathcal{O}(\log n)$
`select_support_sd`	`sd_vector`	64	$\mathcal{O}(1)$

Call `util::init_support(sls,bv)` to initialize `sls` to bitvector `bv`. Call `sls(`i`)` to get `select(`i`)` $= \min\{j \mid$ `rank(`j+1`)` $= i\}$.

Wavelet Trees (WT=BV+RS+SLS)

Wavelet trees represent sequences over byte or integer alphabets of size σ and consist of a tree of BVs. Rank and select on the sequences is reduced to rank and select on BVs, and the runtime is multiplied by a factor in $[H_0, \log \sigma]$.

Class	*Shape*	`lex_ordered`	*Default alphabet*	Travers-able
`wt_rlmn`	underlying WT dependent			×
`wt_huff`	Huffman	×	byte	✓
`wm_int`	Balanced	×	integer	✓
`wt_blcd`	Balanced	✓	byte	✓
`wt_hutu`	Hu-Tucker	✓	byte	✓
`wt_int`	Balanced	✓	integer	✓

Public types: `value_type`, `size_type`, and `node_type` (if WT is traversable). In the following let c be a symbol, i,j,k, and q integers, v a node, and r a range.
Public methods: `size()`, `operator[`i`]`, `rank(`i`,c)`, `select(`i`,c)`, `inverse_select(`i`)`, `begin()`, `end()`.
Traversable WTs provide also: `root()`, `is_leaf(`v`)`, `empty(`v`)`, `sym(`v`)`, `expand(`v`)`, `expand(`v`,`r`)`, `expand(`v`,std::vector<`r`>)`.
`lex_ordered` WTs provide also: `lex_count(`i`,`j`,`c`)` and `lex_smaller_count(`i`,`c`)`. `wt_int` provides: `range_search_2d`.
wt_algorithm.hpp contains the following generic WT method (let wt be a WT object): `intersect(`wt`, vector<`r`>)`, `quantile_freq(`wt`,`i`,`j`,`q`)`, `interval_symbols(`wt`,`i`,`j`,`k`,...)`, `symbol_lte(`wt`,`c`)`, `symbol_gte(`wt`,`c`)`, `restricted_unique_range_values(`wt`,`x_i`,`x_j`,`y_i`,`y_j`)`.

Suffix Arrays (CSA=IV+WT)

Compressed suffix arrays use CIVs or WTs to represent the suffix arrays (SA), its inverse (ISA), BWT, Ψ, and LF. CSAs can be built over byte and integer alphabets.

Class	*Description*
`csa_bitcompressed`	Based on SA and ISA stored in a IV.
`csa_sada`	Based on Ψ stored in a CIV.
`csa_wt`	Based on the BWT stored in a WT.

Public methods: `operator[`i`]`, `size()`, `begin()`, `end()`.
Public members: `isa`, `bwt`, `lf`, `psi`, `text`, `L`, `F`, `C`, `char2comp`, `comp2char`, `sigma`.
Policy classes: alphabet strategy (e.g. `byte_alphabet`, `succinct_byte_alphabet`, `int_alphabet`) and SA sampling strategy (e.g. `sa_order_sa_sampling`, `text_order_sa_sampling`)

Longest Common Prefix (LCP) Arrays

Class	*Description*
`lcp_bitcompressed`	Values in a `int_vector<>`.
`lcp_dac`	Direct accessible codes used.
`lcp_byte`	Small values in a byte; 2 words per large.
`lcp_wt`	Small values in a WT; 1 word per large.
`lcp_vlc`	Values in a vlc_vector.
`lcp_support_sada`	Values stored permuted. CSA needed.
`lcp_support_tree`	Only depths of CST inner nodes stored.
`lcp_support_tree2`	+ large values are sampled using LF.

Public methods: `operator[`i`]`, `size()`, `begin()`, `end()`

Balanced Parentheses Supports (BPS)

We represent a sequence of parentheses as a `bit_vector`. An opening/closing parenthesis corresponds to `1`/`0`.

Class	*Description*
`bp_support_g`	Two-level pioneer structure.
`bp_support_gg`	Multi-level pioneer structure.
`bp_support_sada`	Min-max-tree over excess sequence.

Public methods: `find_open(`i`)`, `find_close(`i`)`, `enclode(`i`)`, `double_enclose(`i`,`j`)`, `excess(`i`)`, `rr_enclose(`i`,`j`)`, `rank(`i`)`[4], `select(`i`)`.
Call `util::init_support(bps,bv)` to initialize a BPS `bps` to `bit_vector bv`.

Suffix Trees (CST=CSA+LCP+BPS)

A CST can be parametrized by any combination of CSA ,LCP, and BPS. The operation of each part can still be accessed through member varaibles. The additional operations are listed below. CSTs can be built for byte or integer alphabets.

Class	*Description*
`cst_sada`	Represents a node as position in BPS. Navigational operations are fast (they are directly translated in BPS operations on the DFS-BPS). Space: $4n + o(n) + \lvert CSA \rvert + \lvert LCP \rvert$ bits.
`cst_sct3`	Represents nodes as intervals. Fast construction, but slower navigational operations. Space: $3n + o(n) + \lvert CSA \rvert + \lvert LCP \rvert$

Public types: `node_type`. In the following let v and w be nodes and i, d, lb, rb integers.
Public methods: `size()`, `nodes()`, `root()`, `begin()`, `end()`, `begin_bottom_up()`, `end_bottom_up`, `size(`v`)`, `is_leaf(`v`)`, `degree(`v`)`, `depth(`v`)`, `node_depth(v)`, `edge(`v`,` d`)`, `lb(`v`)`, `rb(`v`)`, `id(`v`)`, `inv_id(`i`)`, `sn(`v`)`, `select_leaf(`i`)`, `node(`lb`,` rb`)`, `parent(`v`)`, `sibling(`v`)`, `lca(`v`,` w`)`, `select_child(`v`,` i`)`, `child(`v`,` c`)`, `children(`v`)`, `sl(`v`)`, `wl(`v`,` c`)`, `leftmost_leaf(`v`)`, `rightmost_leaf(`v`)`
Public members: `csa`, `lcp`.
The traversal example shows how to use the DFS-iterator.

Range Min/Max Query (RMQ)

A RMQ `rmq` can be used to determine the position of the miniumum value[5] in an arbitrary subrange $[i, j]$ of an preprocessed vector `v`. Operator `operator(`i`,`j`)` returns $x = \min\{r|\ r \in [i,j] \wedge \mathtt{v}[r] \leq \mathtt{v}[k]\ \forall k \in [i,j]\}$

Class	*Space*	*Time*
`rmq_support_sparse_table`	$n \log^2 n$	$\mathcal{O}(1)$
`rmq_succint_sada`	$4n + o(n)$	$\mathcal{O}(1)$
`rmq_succint_sct`	$2n + o(n)$	$\mathcal{O}(1)$

Constructing data structures

Let `o` be a WT-, CSA-, or CST-object. Object `o` is built with `construct(o,file,num_bytes=0)` from a sequence stored in `file`. File is interpreted dependent on the value of `num_bytes`:

Value	*File interpreted as*
`num_bytes=0`	serialized `int_vector<>`.
`num_bytes=1`	byte sequence of length `util::file_size(file)`.
`num_bytes=2`	16-bit word sequence.
`num_bytes=4`	32-bit word sequence.
`num_bytes=8`	64-bit word sequence.
`num_bytes=d`	Parse decimal numbers.

Note: `construct` writes/reads data to/from disk during construction. Accessing disk for small instances is a considerable overhead. `construct_im(o,data,num_bytes=0)` will build `o` using only main memory. Have a look at this handy tool for an example.

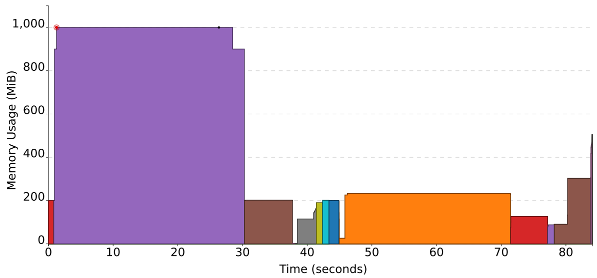
Configuring construction

The locations and names of the intermediate files can be configured by a `cache_config` object. It is constructed by `cache_config(del,tmp_dir,id, map)` where `del` is a boolean variable which specifies if the intermediate files should be deleted after construction, `tmp_dir` is a path to the directory where the intermediate files should be stored, `id` is used as part of the file names, and `map` contains a mapping of keys (e.g. `conf::KEY_BWT`, `conf::KEY_SA`,...) to file paths.
The `cache_config` parameter extends the construction method to: `construct(o,file,config,num_bytes)`.
The following methods (`key` is a key string, `config` represenet a `cache_config` object, and `o` a **sdsl** object) should be handy in customized construction processes:

```
cache_file_name(key,config)
cache_file_exists(key,config)
register_cache_file(key,config)
load_from_cache(o,key,config)
store_to_cache(o,key,config)
```

Resource requirements

Memory: The memory peak of CSA and CST construction occurs during the SA construction, which is 5 times the texts size for byte-alphabets and inputs < 2 GiB (see the Figure below for a 200 MB text) and 9 times for larger inputs. For integer alphabets the construction takes about twice the space of the resulting output.
Time: A CST construction processes at about 2 MB/s. The Figure below shows the resource consumption during the construction of a `cst_sct3<>` CST for 200 MB English text. For a detailed description of the phases click on the figure.



This diagram was generated using the sample program memory-visualization.cpp.

Reading and writing data

Importing data into sdsl structures

`load_vector_from_file(v, file, num_bytes)`
Load `file` into an `int_vector v`. Interpretation of `file` depends on `num_bytes`; see method `construct`.

Store sdsl structures

Use `store_to_file(o, file)` to store an **sdsl** object `o` to `file`. Object `o` can also be serialized into a `std::ostream`-object `out` by the call `o.serialize(out)`.

Load sdsl structures

Use `load_from_file(o, file)` to load an **sdsl** object `o`, which is stored in `file`. Call `o.load(in)` reads `o` from `std::istream`-object `in`.

Utility methods

More useful methods in the `sdsl::util` namespace:

Method	*Description*
`pid()`	Id of current process.
`id()`	Get unique id inside the process.
`basename(p)`	Get filename part of a path `p`.
`dirname(p)`	Get directory part of a path `p`.
`demangle(o)`	Demangles output of `typeid(o).name()`.
`demangle2(o)`	Simplifies output of `demangle`. E.g. removes `sdsl::`-prefixes, ...
`to_string(o)`	Transform object `o` to a string.
`assign(o1,o2)`	Assign `o1` to `o2`, or swap `o1` and `o2` if the objects are of the same type.
`clear(o)`	Set `o` to the empty object.

Measuring and Visualizing Space

`size_in_bytes(o)` returns the space used by an **sdsl** object `o`. Call `write_structure<JSON_FORMAT>(o,out)` to get a detailed space breakdown written in JSON format to stream `out`. `<HTML_FORMAT>` will write a HTML page (like this), which includes an interactive SVG-figure.

Methods on words

Class `bits` contains various fast methods on a 64-bit word x. Here the most important ones.

Method	*Description*
`bits::cnt(`x`)`	Number of set bits in x.
`bits::sel(`x`,`i`)`	Position of i-th set bit, $i \in [0, \mathtt{cnt}(x) - 1)$.
`bits::lo(`x`)`	Position of least significant set bit.
`bits::hi(`x`)`	Position of most significant set bit.

Note: Positions in x start at 0. `lo` and `hi` return 0 for $x = 0$.

Tests

A `make test` call in the test directory, downloads test inputs, compiles tests, and executes them.

Benchmarks

Directory benchmark contains configurable benchmarks for various data structure, like WTs, CSAs/FM-indexes (measuring time and space for operations count, locate, and extract).

Debugging

You get the gdb command `pv <int_vector> <idx1> <idx2>`, which displays the elements of an `int_vector` in the range `[idx1, idx2]` by appending the file sdsl.gdb to your `.gdbinit`.

Notes

1 `select_0_type` not defined for `sd_vector`.
2 It is also possible to rank `0` or the patterns `10` and `01`.
3 It is also possible to select `0` or the patterns `10` and `01`.
4 For PBS the bits are counted in the prefix $[0..i]$.
5 Or maximum value; can be set by a template parameter.